UNIVERSITY OF APPLIED SCIENCES
HOCHSCHULE MITTWEIDA

Angewandte Computer- und Biowissenschaften

Professur Medieninformatik

Bachelorarbeit

Entwurf und Implementierung einer dynamischen Softwareplattform für konfigurier- und erweiterbare Simulationen auf Basis der Unity Engine

Gabor Schulze

Mittweida, den 14. Dezember 2017

Erstprüfer: Prof. Dr. rer. nat. Marc Ritter
Zweitprüfer: Prof. Dr. rer. nat. Petra Radehaus
Co-Betreuer: Manuel Heinzig, M. Sc.

Schulze, Gabor
Entwurf und Implementierung einer dynamischen Softwareplattform für konfigurier- und erweiterbare Simulationen auf Basis der Unity Engine
Bachelorarbeit, Angewandte Computer- und Biowissenschaften
Hochschule Mittweida– University of Applied Sciences, Dezember 2017

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einführung und Motivation

1.1. Motivation

Virtual Reality bietet aufgrund von hoher Immersion und präziser Steuerung (Quelle /Beleg) mittels handlicher Bewegungen neue Möglichkeiten im Bereich der Lernsoftware und Simulatoren. Simulatoren, in welchen Arbeitsabläufe nachgeahmt werden, sparen Zeit und Kosten, da keine realen Materialien und Anweiser nötig sind. Arbeitsabläufe, bei denen Geschick und Präzision der Hände gefragt sind, können je nach Komplexität im virtuellen Raum umgesetzt werden. Das Pipettieren ist ein solcher Vorgang. Die Simulation des Pipettiervorgangs, sowie diese Arbeit sind Projekte der Task Force *BIOTECMI* [Kru17,]. Beim Pipettieren ist es möglich, viele Fehler zu begehen, wenn die Routine fehlt. In einem virtuellen Raum können wesentliche Grundkenntnisse erlernt und eine gewisse Geschicklichkeit eingeübt werden. Anhand der Definition von Regeln, Abläufen und Abhängigkeiten lassen sich solche Abläufe konfigurieren. Diese Grundlage lässt es zu einen Softwarekern konstruieren, der für verschiedene Simulationsszenarien geeignet ist.

1.2. Aufgabenstellung

Im Rahmen der Arbeit soll an der Hochschule Mittweida eine Virtual Reality-Simulationssoftware entstehen, welche aus einen dynamischen Softwarekern besteht. Dieser soll durch das dynamische Laden von modularen Bestandteilen, wie Geometriedaten und Verhaltensweisen, sowie den Einsatz von Konfigurationsdateien für weitere Anwendungsfälle und Szenarien erweiterbar sein. Es existieren eine Vielzahl an Entwurfsmustern, die für die Erstellung von Software mit derart dynamischen Prozessen geeignet sind. Für die Arbeit ist die Eingrenzung und Auswahl passender Entwurfsmuster eine wichtige Aufgabe. Es gilt zu planen, wie die Objekte mit Schnittstellen versehen werden, die eine solche Dynamik unterstützt.

Ein Katalog von Verhaltensweisen, die als Grundlage für die agierenden Objekte dienen, soll erstellt werden und für weitere Verhaltensweisen, sowie für Erweiterungen offen sein. Objekte sollen über Konfigurationsdateien vor dem Start festgelegt werden können. Ein konfigurierbarer Ablauf soll definiert und so umgesetzt werden, dass Start-, Fehler- und Endbedingungen festlegbar sind. Startbedingungen beinhalten auch die Festlegung zum Einsatz kommender Objekte.

Der konkrete Anwendungsfall für das Pipettieren soll durchdacht und mit passenden Verhaltensweisen umgesetzt werden. Die Implementierung erfolgt innerhalb der Unity-Engine. Für die Umsetzung des Anwendungsfalles Pipettieren ist der Vorgang des Pipettierens zu erarbeiten.

1.3. Ziel der Arbeit

Ziel der Arbeit soll es sein, in Kooperation mit Kerstin Knura (Hochschule Mittweida) für die Hochschule Mittweida eine VR-Laborumgebung zu erstellen, in der Pipettierübungen ausgeführt werden können. Diese Übungen werden mittels der VR-Brille HTC Vive und den dazugehörigen Controllern ausgeführt. Die Übungsszenarien sollen durch Konfigurationsdateien eingerichtet werden. Ausgangsobjekte mit ihren Abhängigkeiten, Verhaltensweisen der Objekte und Fehler sowie Endbedingungen sollen wählbar sein. Ein erweiterbarer Katalog an Modulen aus Geometriedaten und Verhaltensweisen für Objekte dient hierzu als Grundlage. Dadurch ist die entstandene Software dynamisch für neue Simulationsfälle erweiterbar. Diese modulare und dynamische Struktur bildet damit auch die Grundlage für Szenarien, die sich nicht mit dem Pipettieren befassen. Erweiterungen sind nicht gezwungen, die VR-Methoden zu nutzen.

1.4. Kapitelübersicht

Diese Bachelorarbeit befasst sich mit der Entwicklung einer Softwareplattform, sowie deren Verwendung für die Anwendung des Pipettierens. Kapitel 2 befasst sich mit den Grundlagen, welche für den Aufbau und die Funktionsweisen in Betracht gezogen wurden. Dies umfasst auch das Thema Virtual Reality, welches eine fundamentale Rolle spielt und die damit eng verbundenen Engines. Erläuterungen ausgewählter

Praktiken der Informatik sowie eine Betrachtung des Pipettiervorgangs werden analysiert. Auf dieser Grundlage schließt sich das Kapitel 3 an. Hier wird darauf eingegangen, wie die Software und ihre Bestandteile Aufgebaut sind und unter welchen Gesichtspunkten diese Entscheidungen getroffen werden. Darauf aufbauend widmet sich das Kapitel 4 der Implementierung der Anwendung. Konkrete Vorgänge und Zusammenhänge werden veranschaulicht und anschließend im Kapitel 5 ausgewertet. Eine Zusammenfassung und Ausblick über Zukünftige Anknüpfungspunkte und Verbesserungen im Kapitel 6 rundet diese Arbeit ab.

2. Grundlagen

Im Anschluss an das einleitende Kapitel 1 folgen die Grundlagen für das Verständnis des entworfenen Systems. Ein Einblick in die objektorientierte Programmierung und übliche Programmierpraktiken dynamischer Systeme bildet die Grundlage für die Planung und Architektur von Softwaresystemen. Um den Anspruch der Konfigurierbarkeit zu wahren, erfolgt eine Analyse von Datenhaltungspraktiken und Formaten. Da die Umsetzung des Systems mittels einer Game Engine erfolgt, gilt es, solche auf definierte Anforderungen zu untersuchen. Anschließend erfolgt die Ermittlung von Verfahren für benutzerfreundliche Abläufe, mit dem Hintergrund, das System auch für Laien bedienbar zu gestalten. Das Grundlagenkapitel endet mit einem Überblick des Pipettierablaufes, sowie mit Fehlern, die im Verlauf auftreten können.

2.1. Virtuelle Realität

Die Umsetzung der Anwendung für das Pipettieren findet für eine Virtual Reality Brille statt. Virtuelle Realität wird häufig mit VR abgekürzt. Laut Burdea [Bur03, S. 2ff] ist virtuelle Realität eine Simulation, in welcher durch Computergrafiken eine realistisch wirkende Welt erzeugt wird. Diese erzeugte Welt ist nicht statisch und kann durch Benutzereingaben verändert werden. Die Schlüsselfunktion ist hierbei die Echtzeitinteraktion, welche eine erhöhte Immersion herbeiführt.

Als Virtual Reality-Hardware kommt die VR Brille HTC Vive zum Einsatz. Die HTC Vive verfügt laut Herstellerangaben [VIV17,] über eine Displayauflösung von insgesamt 2160×1200 Pixel. Das bedeutet eine Auflösung von 1080×1200 Pixel pro Auge. Das abgedeckte Sichtfeld beträgt 110 Grad. Über HDMI- und USB-Verbindungen findet die Kommunikation der HTC Vive mit einem Computer satt. Zwei Tracker unterstützen die Positionserkennung der VR-Brille und der zugehörigen Controller.

2.2. Game Engines

Eine Game Engine ist ein Programm, das bei der Entwicklung von Anwendungen eine Hilfe bietet. Sie übernimmt Prozesse in der Entwicklung einer Anwendung, sowie in ihrer Ausführung. Game Engine bedeutet wörtlich übersetzt "Spiele Motor" und wird auch kurz Engine genannt. Sie sind aus der Spieleentwicklung heraus entstanden und nicht immer klar von der Software des Spieles unterscheidbar. Es existieren Engines, die entworfen wurden, um Spiele zu entwickeln und jene, die selbst aus Spielen heraus entstehen. Der signifikanteste Unterschied zwischen einer Game Engine und einem Spiel, das sich nicht als Game Engine kategorisieren lässt, ist die Änder- und Erweiterbarkeit des Inhaltes. Game Engines werden im Allgemeinen mit Focus auf spezielle Genres und Hardwareplattformen entwickelt. Hintergrund dessen ist, dass es sehr aufwendig ist, eine Engine zu implementieren, die auf alle Genres und Hardwareplattformen abgestimmt ist. Grund dafür ist, dass Spiele die mittels einer Game Engine entworfen werden, sehr spezielle Ansprüche besitzen. Für eine optimale Umsetzung eines Spieles ist es erforderlich, dass die zum Einsatz kommende Engine auf dieses abgestimmt ist. Wenn eine Engine für mehrere Anwendungen, welche auf unterschiedlichen Einsatzgebiete abzielen (zum Beispiel Genre oder Hardwareplattform), entworfen wird, folgen zwangsläufig Abwägungen in Design und Implementierung, da die resultierenden Anforderungen durchaus gegensätzlicher Natur sein können. [Gre14, S. 11ff.]

Jede Engine bietet unterschiedliche Funktionen und Eigenschaften an. So kann eine Engine beispielsweise die Berechnung von physikalischen Verhaltensweisen übernehmen, ohne dass Anwender diese selber implementieren müssen. Es existiert eine Vielzahl von unterschiedlichen Engines für verschiedenste Anwendungsfälle. Game Engines können für die Entwicklung von Anwendungen im dreidimensionalen Raum hilfreich sein. Essentielle Bestandteile wie die Kommunikation mit der Grafikeinheit eines Computers, Berechnung von physikalischen Verhaltensweisen und das Erstellen einer ausführbaren Datei sind große Mengen an Implementierungsaufwand, die durch das Einbeziehen einer Game Engine wegfallen.

2.2.1. Game Engines im Vergleich

Im Abschnitt 3.1 werden die Anforderungen an die zu verwendende Game Engine herausgearbeitet. im Vorfeld dazu findet nachfolgend der Vergleich von ausgewählten

Features	Unity Engine	Unreal Engine	Cry Engine
Vive Kompatibel	Ja	Ja	Ja
Physik Simulation	Ja	Ja	Ja
Renderingverwaltung	Ja	Ja	Ja
Skript unterstützung	C# und Javascript	C++ und das visuelle Skriptsystem Blueprint	C++, C# und Lua
Inhalte laden	Bereitstellung von Assetbundles (siehe 2.2.2)	Implementierung notwendig	Implementierung notwendig

Tabelle 2.1.: Übersicht Game Engines [Cry17, Uni17, Epi17]

Game Engines statt. Bei den Engines handelt es sich um die Unity Engine, die Unreal Engine und die Cry Engine. Tabelle 2.1 enthält die laut Online Handbüchern der jeweiligen Engine unterstützten Funktionen. Gesichtspunkte sind die in 3.1 herausgestellten Anforderungen an die Engines.

2.2.2. Die Unity Engine

Der Abschnitt 3.2 beinhaltet die Auswertung der zur Wahl stehenden Engines. Das Resultat der Auswertung ist, dass die Unity Engine am besten in das Aufgabenfeld passt. Folglich kommt die Unity Engine zum Einsatz. Bei der im Rahmen dieser Arbeit verwendeten Version handelt es sich um die Version 2017.1.

Die Unity Engine ist eine von Unity Technologies entwickelte Software, welche als Game Engine klassifiziert werden kann. Sie ist konzipiert, um Anwendungen, vorrangig Spiele, zu entwickeln. [Uni17,]

Eine Erläuterung über die Funktionsweise und den Aufbau findet im Folgenden statt. Der Fokus liegt hierbei auf Funktionsweisen und Elemente, die für die Umsetzung der in dieser Arbeit behandelten Themengebiete erforderlich sind. Die Beschreibung des kompletten Aufbaus der Engine würde über den Rahmen dieser Arbeit deutlich hinausgehen.

Aufbau Jede Entwicklung innerhalb der Unity Engine findet in einem Projekt statt. Ein Projekt ist die Sammlung aller Komponenten, die für eine zu entwickelnde Anwendung genutzt werden [Uni17,]. Die Unity Engine verfügt über die Funktionalität ein Projekt zu einer lauffähigen Anwendung umzuwandeln. Bei dem Resultat handelt es sich um ein Build [Uni17,].

Assets Ein Asset stellt die Repräsentation eines jeden Objektes dar, welches nativ in einem Unity-Projekt genutzt werden kann. Beispiele hierfür sind 3D-Modelle, Audiodateien oder Bilddateien. [Uni17,]

GameObjects GameObjects sind grundlegende Objekte in der Unity Engine. Sie repräsentieren Akteure, Requisiten und Landschaften. GameObjects führen selber keine Aktionen aus, sind aber Behälter für Componenets (Komponenten). [Uni17,]

Components Components repräsentieren Verhaltensweisen und sind dem funktionalen Teil eines jeden GameObjects verantwortlich. Beispiele für Components sind unter anderem Renderer, Skripte, Positionen im Raum und physikalisches Verhalten. [Uni17,]

Skripts Skripts steuern den Ablauf einer auf der Unity Engine-basierten Anwendung. Sie sind ein Bestandteil der vom Nutzer geschaffenen Funktionalität. Über Skripts findet unter anderem die Regelung von Reaktionen auf Nutzereingaben statt. Die Unity Engine unterstützt das Skripten mit den Programmiersprachen C# und JavaScript. Eine Schnittstelle (API) ermöglicht die Nutzung von Funktionen, die die Unity Engine bereitstellt und Zugriff auf Kernelemente wie beispielsweise Components ermöglicht. [Uni17,]

AssetBundles AssetBundles sind Archive, welche Assets beinhalten. Diese verfügen über die Eigenschaft nativ zur Laufzeit der Anwendung geladen zu werden. Für die Verwendung von AssetBundles ist es nicht erforderlich, dass diese ihren Ursprung im selben Projekt haben. AssetBundles werden mit der Unity Engine ohne Nutzung von dritten Programmen erzeugt. Die Unity Engine stellt eine Funktion bereit, um

AssetBundles zu laden. Dabei wird das AssetBundle mit seinem Pfad im Dateisystem angegeben.

MonoBehaviour MonoBehaviour ist eine Klasse, die Schnittstellen für den Ablauf und interne Funktionen anbietet. Eine Klasse, die von MonoBehaviour erbt, hat beispielsweise Zugriff auf die Update-Funktion. Die Engine ruft den in der Update-Funktion geschriebenen Code in zeitlichen Intervallen auf, unter der Bedingung, dass die Klasse instanziiert und als Component an ein GameObject geheftet ist.

2.3. Datenhaltung

Die Datenhaltung ist ein essentieller Bestandteil der zu entwickelnden Anwendung. Dieser Abschnitt widmet sich der Datenhaltungsansätze, sowie der Umsetzungsmöglichkeiten durch verschieden Formate. Konfigurierbare Ladevorgänge, wie sie innerhalb dieser Arbeit entstehen sollen, verlangen eine Struktur, die es erlaubt, Textinformationen anzugeben, um verschiedene Definitionen vorzunehmen. Beispiel eines solchen Verfahrens innerhalb dieser Arbeit ist die Adressierung eines AssetBundles (siehe 2.2.2).

Laut Ried und Rothfuss [RR03, S. 11] existieren verschiedene Strukturansätze, um mit der Datenhaltung umzugehen. Traditionelle Datenbanken beherbergen Datensätze, die gleich aufgebaut sind (strukturierte Daten). Als Gegenentwurf existieren die semistrukturierten Daten, die keine oder eine hierarchische Struktur besitzen.

Das XML-Format XML (Extensible Markup Language) ist eine Auszeichnungssprache, die entwickelt wurde, um strukturierte Textdokumente zu erzeugen, die vom Menschen lesbar sind. [RR03, S. 231]

Der Inhalt einer gültigen XML-Datei ist als Dokument bezeichnet [RR03, S. 41]. Ein XML-Dokument besteht aus Elementen, deren Beginn und Ende durch Tags (Beginn : <BeispielTag> Ende : <\BeispielTag>) gekennzeichnet wird. Ein Dokument besitzt ein Wurzelelement, das die oberste Ebene der Hierarchie darstellt. Elemente innerhalb eines XML-Dokuments sind in einer Baumstruktur angeordnet. Ein Element ist ein Knoten innerhalb dieser Baumstruktur. Inhalt eines Elements sind entweder ein Wert oder Unterknoten (Knoten, welche diesem in der Hierarchie

untergeordnet sind, auch Kindknoten genannt). Zu beachten ist, dass ein Element mehrere Knoten unter sich haben kann. Diese befinden sich zwischen dem Start- und Ende-Tags des Elternknotens (Knoten, die diesem in der Hierarchie übergeordnet sind). Ein Wert ist eine Zeichenkette. [Jos08, S. 5ff.]

Diese XML-Konventionen erlauben das Speichern von Zeichenketten in einer hierarchisch strukturierten Form. Innerhalb dieser Baumstruktur ist es möglich, Listen anzulegen (mehrere Elemente mit dem gleichen Tag werden einem Elternknoten zugeordnet).

2.4. Programmierparadigmen

In der Disziplin der Programmierung existieren zahlreiche Praktiken, die sich über den Verlauf der Zeit etabliert haben. Je nach Anwendungsgebiet und Plattform haben sich zahlreiche Programmiersprachen, Konventionen und Muster herausgebildet. Dieser Abschnitt widmet sich vorrangig der Objektorientierten Programmierung und der in ihr gebildeten Entwurfsmuster.

2.4.1. Objekt Orientierte Programmierung

> „Die Objektorientierung heißt Objektorientierung, weil diese Methode die in der realen Welt vorkommenden Gegenstände als Objekte ansieht.“ [Oes01, S, 37].

> „[R]eal vorkommende Gegenstände werden auf wenige, in der Situation bedeutsamen Eigenschaften reduziert.“ [Oes01, S. 38].

Objektorientierte Programmierung, oder auch kurz OOP, ist also ein Programierparadigma, welches der Idee zu Grunde liegt, Software so zu strukturieren, dass sie sich am Aufbau der Realität orientiert. Das fundamentale Konzept der OOP wird im Folgenden beschrieben.

Klassen, Objekte und Instanzen Klassen, Objekte und Instanzen sind die Grundbausteine der OOP. Das Verhältnis dieser drei Begriffe zueinander ist für das Verständnis der OOP wichtig.

> „Eine Klasse beschreibt die Struktur und das Verhalten einer Menge gleichartiger Objekte. Ein Objekt ist eine zur Ausführungszeit vorhandene und für ihre Instanzvariablen Speicher allokierende Instanz, die sich entscheidend dem Protokoll ihrer Klasse verhält.“ [Oes01, S. 38].

> „Eine Klasse ist die Definition der Attribute, Operationen und der Semantik für eine Menge von Objekten. Alle Objekte einer Klasse entsprechen dieser Definition.“ [Oes01, S. 209].

Aus diesem Zusammenhang geht folgendes hervor: Eine Klasse ist eine Beschreibung, die Variablen deklariert und eine Verhaltensweise (Funktionen) festlegt. Ein Objekt ist etwas, das nach dem Bauplan einer Klasse angelegt (instanziiert) wird und auf einen eigenen Speicher zugreift. Objekte haben im Gegensatz zu Klassen tatsächlich die Möglichkeit, zu agieren.

Ein Objekt, das nach Bauplan einer Klasse angelegt wurde, wird auch Instanz oder Exemplar dieser Klasse genannt.[Oes01, S. 39]

> „Ein Objekt kann die in der Klasse definierten Nachrichten empfangen, d.h. es besitzt für jede Nachricht eine entsprechende Operation.“ [Oes01, S. 217].

Objekte haben also die Möglichkeit, mit anderen Objekten über definierte Operationen zu kommunizieren und Daten auszutauschen, beziehungsweise weiterzuleiten.

Attribute und Operationen

> „Attribute beschreiben die Struktur der Objekte: ihre Bestandteile und die in ihnen enthaltenen Informationen bzw. Daten.“ [Oes01, S. 40]

> „Ein Attribut ist ein (Daten-) Element, das in jedem Objekt einer Klasse gleichermaßen enthalten ist und von jedem Objekt mit einem individuellen Wert repräsentiert wird.“ [Oes01, S. 219].

Attribute sind also die tatsächlichen Daten (Variablen), die ein Objekt ausmachen. Die Gesamtheit der Attribute und ihrer Werte können auch als Zustand eines Objektes bezeichnet werden. Aus dem Vorangegangen geht hervor, dass Attribute in ihrem Wert veränderbar sind, also unterschiedliche Werte beziehungsweise Daten annehmen können.

> „Operationen beschreiben das Verhalten der Objekte.“ [Oes01, S. 40]

Operationen führen Berechnungen eines Objektes aus. Als Verhalten eines Objektes haben sie Zugriff auf die Attribute. Mittels Attributen und Parametern erhalten Operationen Informationen (Werte oder auch Daten), mit denen sie verfahren können. Operationen werden auch Methoden oder Funktionen genannt.

> „Operationen sind Dienstleistungen, die von einem Objekt angefordert werden können, ...“ [Oes01, S. 222]

Operationen werden also angefordert und anschließend ausgeführt. Wichtig ist, dass sie von einem Objekt angefordert werden. Das beinhaltet auch das Objekt, in welchem die Operation ausgeführt wird, aber auch andere Objekte. Operationen können über Parameter (Eingabewerte) und Rückgabewerte verfügen. [Oes01, S. 222]

Der Konstruktor

> „Wenn eine Klasse oder Struktur erstellt wird, wird deren Konstruktor aufgerufen. Konstruktoren haben den gleichen Namen wie die Klasse oder Struktur, und sie initialisieren, normalerweise die Datenmember des neuen Objekts.“ [Mic17,]

Vererbung Vererbung ist die Beziehung einer Ober- und einer Unterklasse zueinander. Attribute und Operationen einer Oberklasse werden für eine Unterklasse zugänglich gemacht. [Oes01, S. 257]

Kapselung

> „Das Kapselungsprinzip
>
> Klassen fassen Attribute und Operationen zu einer Einheit zusammen. Attribute sind nur indirekt über die Operation der Klasse zugänglich.“ [Oes01, S. 40]

Attribute sind für objektfremde Operationen nicht erreichbar. Ermöglicht wird dieser Zustand durch die Sichtbarkeit eines Attributes. Somit haben nur Operationen die Möglichkeit, auf Attribute zuzugreifen.

Sichtbarkeit Sichtbarkeiten sind je nach verwendeter Programmiersprache unterschiedlich. Sichtbarkeit gibt an, welche Klassen beziehungsweise Objekte auf ein Attribut zugreifen können. Drei bekannte Sichtbarkeitskennzeichen sind public (öffentlich), private (privat) und protected (geschützt). Public bedeutet, dass ein Attribut nach außen hin sichtbar und damit für alle benutz- und veränderbar ist. Auf protected-Attribute haben nur die Klasse selbst und Unterklassen Zugriff. Private Attribute sind nur von der Klasse selbst zu erreichen. [Oes01, S. 220ff.]

Abstraktion

> „Von einer abstrakten Klasse werden niemals Exemplare erzeugt; sie ist bewusst unvollständig und bildet somit die Basis für weitere Unterklassen, die Exemplare haben können. “ [Oes01, S. 214]

In abstrakten Klassen können abstrakte Funktionen definiert werden. Die Implementierung dieser findet aber erst in den erbenden Unterklassen statt. [Oes01, S. 214]

Abstraktion ist auch durch Interfaces (engl. Schnittstellen) möglich. Diese sind abstrakte Klassen, die ausschließlich abstrakte Operationen definieren [Oes01, S. 228]. Die Definition von Attributen ist in Interfaces nicht möglich.

> „Eine gewöhnliche Klasse kann mehrere Schnittstellen implementieren und darüber hinaus weitere Eigenschaften erhalten. “ [Oes01, S. 228]

Klassenattribute Klassenattribute (auch Klassenvariablen genannt) sind Attribute, welche nicht einer Instanz, sondern allen Instanzen der Klasse gehören. Das heißt, dass alle Objekte der gleichen Klasse Zugriff auf ein gemeinsames Attribut besitzen [Oes01, S. 220]. In C# findet die Deklaration einer Klassenvariable durch das Schlüsselwort static statt [Mic17,].

2.4.2. Entwurfsmuster

Die Implementierung von Entwurfsmustern ist eine empfehlenswerte Vorgehensweise für Entwickler von objektorientierten Softwaresystemen, da sie die Vorteile aus folgender Erörterung mit sich bringt. Nach Gamma et al. [GHJV15, Seite 17] sind Entwurfsmuster bereits entwickelte und bewährte Problemlösungen im Gebiet der objektorientierten Programmierung. Sie beschreiben Herangehensweisen für häufig auftretende Probleme, an welchen schon viele Entwickler gearbeitet haben. Diese Entwurfsmuster wurden mit Hinblick auf Wiederverwendbarkeit und Flexibilität konstruiert.

Die Umsetzung der zu entwickelnden Software erfolgt durch die objektorientierte Programmiersprache C# (siehe Kapitel 4), welche damit grundlegende, für Entwurfsmuster benötige Funktionen unterstützt.

Für die Nutzung von Entwurfsmustern findet im Folgenden ein Überblick über einige Entwurfsmuster statt, wobei besonders auf Erzeugungs- und Verhaltensmuster eingegangen wird.

Erzeugungsmuster

Entwurfsmuster, die als Erzeugungsmuster klassifiziert werden, übernehmen die Aufgabe, den Instanziierungsprozess zu abstrahieren. Klassenbasierte Erzeugungsmuster bedienen sich dabei des Vererbungsprinzips. Objektbasierte Erzeugungsmuster leiten die Aufgabe der Instanziierung an andere Objekte weiter.[GHJV15, Seite 123]

> „Erzeugungsmuster folgen zwei stets wiederkehrenden Leitmotiven: Zum einen kapseln sie die Informationen zu den vom System verwendeten konkreten Klassen, und zum anderen verbergen sie die Art und Weise,

> wie die Instanzen dieser Klassen erzeugt und zusammengeführt werden.“ [GHJV15, Seite 40]

Das Singleton (dt. Einzelstück) Bei dem Singleton-Muster handelt es sich um ein objektbasiertes Erzeugungsmuster [GHJV15, S. 172]. Sein Zweck wird im Folgenden wiedergegeben.

> „Sicherstellung der Existenz nur einer einzigen Klasseninstanz sowie Bereitstellung eines globalen Zugriffspunkts für diese Instanz.“ [GHJV15, Seite 172]

Implementierung Das Singleton Muster sorgt folgendermaßen für die Umsetzung. Eine Singletonklasse besitzt eine Variable vom Typ ihrer eigenen Klasse. Diese Variable ist static (siehe 2.4.1). Klassen, die die Instanz der Singletonklasse erfahren möchten, erhalten diese Variable als Referenz zurück. Durch Ändern der Sichtbarkeit des Konstruktors ist die Instanziierung nur durch die Singletonklasse selbst möglich. Das ist durch die Implementierung einer static-Funktion in der Singletonklasse möglich. Diese prüft nach, ob die static-Variable vom Typ der Singletonklasse bereits referenziert ist, falls dies nicht der Fall ist, so referenziert die Klasse sich selber, andernfalls wird sie gelöscht. [GHJV15, Seite 172]

Die Factory Method (dt. Fabrik Methode) Bei dem Factory-Muster handelt es sich um ein klassenbasiertes Erzeugungsmuster [Gol14, S. 244]. Sein Zweck wird im Folgenden wiedergegeben.

> „Eine Klasse soll ein Objekt erzeugen, dessen Typ sie nicht kennt bzw. dessen Typ erst zur Laufzeit des Programms bekannt ist. Die erzeugende Klasse kennt nur die Basisklasse des zu erzeugenden Objekts zur Kompilierzeit, weiß aber nicht, von welcher Unterklasse das entsprechende Objekt zur Laufzeit im lauffähigen Programm später sein soll.“ [Gol14, S. 243]

nach Gamma et al. [GHJV15, S. 154] gibt es Zwei Varianten dieses Musters. In einer ist die Klasse, welche die Instanziierungen ausführt, abstrakt und in der anderen nicht. Im Folgenden findet die Erläuterung für den zweiten Fall ohne abstrakte Erzeugerklasse statt.

Implementierung Die Factory-Methode erzeugt Instanzen einer abstrakten Produkt-Klasse. Diese muss samt abstrakter Funktionen von jeder durch die Factory-Methode erzeugten konkreten Klassen implementiert werden. Die erzeugende Methode benötigt den Klassentyp des zu instanziierenden Objektes. Anhand des Typs kann die Methode die benötigte konkrete Klasse instanziieren. [GHJV15, S. 154ff.]

Verhaltensmuster

> „Verhaltensmuster beschäftigen sich schwerpunktmäßig mit Algorithmen und der Zuweisung von Zuständigkeiten an Objekte. Sie beschreiben nicht nur Patterns von Objekten und Klassen, sondern auch deren wechselseitige Kommunikationsmuster.“ [GHJV15, S. 279].

Klassenbasierte Verhaltensmuster wenden das Vererbungsprinzip für die Verhaltenszuordnung an. Objektbasierte Verhaltensmuster hingegen machen sich die Objektkomposition zunutze. [GHJV15, S. 279]

Strategy (dt. Strategie) Muster Das Strategy-Muster ist ein objektbasiertes Verhaltensmuster [Gol14, S. 172]. Sein Zweck lautet wie folgt:

> „Das Strategie-Muster soll erlauben, dass ein ganzer Algorithmus in Form einer Kapsel ausgetauscht wird.“ [Gol14, S. 174]

Implementierung Verschiedene Algorithmen, die miteinander austauschbar sein sollen, implementieren das gleiche Interface (siehe 2.4.1) in einer Klasse. Die Algorithmen werden in Klassen gekapselt. Jeder dieser Klassen kann anstelle einer anderen mit dem verwendeten Interface ausgetauscht werden. Dadurch kann ein Objekt zur Laufzeit sein Verhalten ändern. [Gol14, S. 174]

2.5. Usability

Usability bedeutet übersetzt Verwendbarkeit und bezieht sich auf Hilfestellungen für den Anwender. Diese Hilfestellungen sorgen dafür, dass Nutzer einer Software bei der Bedienung unterstützt werden.

Hilfestellungen werden in der Regel über Feedbacks (Rückmeldungen) gegeben. Beispielsweise ist die Munitionsanzeige in einem Ego-Shooter ein Feedback. Es zeigt Informationen an, welche sich durch das Verhalten des Spielers ändern können. Wenn zu viele solcher Feedbacks existieren leidet der Nutzer, da er die Übersicht verliert. Es gilt, nicht inflationär mit Rückmeldungen umzugehen und diese, falls sie visueller Natur sind, in unterschiedlichen Regionen des Bildschirms anzuzeigen, im Idealfall an den Stellen, die für ihr auftreten verantwortlich sind. Ein Beispiel wäre die Bewegung des Mauszeigers über ein Objekt, das sich infolgedessen verfärbt und somit eine visuelle Rückmeldung erteilt. [Moo11, Seite 321 ff.]

nach Moore [Moo11, Seite 329 ff.] existieren verschiedene Arten von Rückmeldungen. In Spielen passiert es häufig, dass dazu visuelle, akustische und haptischen Reize erzeugt werden. Feedbacks sollen vor allem dem Nutzer mitteilen, ob er sich gerade richtig oder falsch verhalten hat. Dies ermöglicht dem Nutzer aus seiner Handlungsweise zu lernen. Dabei gilt es zu beachten, dass es bereits angewendete Verfahren aus schon bestehenden Anwendungen gibt und Nutzer sich an diese gewöhnt haben. Das Einbringen neuer Verfahrensweisen könnte dazu führen, dass Anwender sich erst an diese gewöhnen müssen. Daher ist es prinzipiell ratsam, sich an bestehenden Verfahren zu orientieren.

2.6. Der Pipettiervorgang

Da die entstehende Software das Pipettieren in einem VR-Labor umsetzt, ist es unumgänglich, sich mit der Praxis des Pipettierens zu befassen. Die Umsetzung des Pipettierens findet durch Kerstin Knura (Hochschule Mittweida) statt. Dabei kommen Mikroliterpipetten zum Einsatz [Kru17,].

2.6.1. Grundlagen des Pipettierens

Mikroliterpipetten verfügen über auswechselbare Spitzen. An den Pipetten befindet sich ein Einstellrad, durch welches das Volumen geregelt wird. Zunächst wird an der Pipette eine Spitze angebracht. Anschließend wird der Aufnahmeknopf gedrückt bis ein Widerstand merkbar ist. Die Spitze der Pipette wird nun in die Flüssigkeit eingetaucht. Die Pipette ist dabei senkrecht zu halten. Das langsame Nachgeben

des Aufnahmeknopfes, bis in die Ausgangsposition, führt nun zur Aspiration. Die Spitze muss nun an die Wand des Gefäßes gehalten werden, wo sie durch erneutes langsames Drücken des Aufnahmeknopfes die Flüssigkeit wieder abgibt. Eine Spitze wird über einen Abwurfknopf von der Pipette entfernt. Eine Spitze ist nicht für verschiedene Flüssigkeiten zu nutzen, da sonst Verunreinigungen entstehen. [Kru17,]

2.6.2. Umsetzung des Pipettierens

Das System wird für den Anwendungsfall des Pipettierens umgesetzt. Es entsteht ein Labor, in welchem Pipetten und zugehörige Spitzen zur Verfügung stehen. Gefäße mit und ohne Flüssigkeit sind vorhanden. Durch das Pipettieren ist es möglich, Flüssigkeit in leere Gefäße zu füllen. [Kru17,]

Die zum Einsatz kommenden 3D-Modelle sind durch Kerstin Knura entstanden. Die Funktionalität des Pipettiervorgangs ist durch Kerstin Knura realisiert worden. [Kru17,] Es gibt jedoch Schnittpunkte, ein solcher Fall ist die Funktionalität der Pipetten. Das Verhalten dieser ist eine Kooperation.

3. Konzeptionierung und Spezifikation

Aufbauend auf den Grundlagen vom Kapitel 2 erfolgt die Festlegung von Anforderungen an die Software sowie die Begründung ihres Aufbaus.

3.1. Anforderungsanalyse an die Game Engine

Im Kapitel 2.2 wird erläutert, dass die Wahl der Engine in Hinsicht auf Plattform und Genre fundamental für die mit ihr entwickelten Anwendung ist. Dieser Abschnitt befasst sich mit den Kriterien und Ansprüchen, welche an eine Game Engine gestellt werden, damit diese für die Entwicklung des Toolkits und des Virtual Reality-Labors geeignet ist. Folgende Schwerpunkte sind an die Game Engine zu Stellen.

HTC Vive-Kompabilität Als Virtual Reality-Hardware kommt die VR-Brille HTC Vive zum Einsatz. Damit diese effizient und ohne zusätzlichen Implementierungsaufwand genutzt werden kann, ist eine eingebaute Unterstützung der HTC Vive erforderlich. Diese Unterstützung beinhaltet das Tracking der Brille sowie der Controller. Eine Darstellung der Controller und Erkennung der Blickrichtung des Nutzers sind ebenfalls erforderlich.

Physik Simulation Die Simulation von physikalischen Prozessen, genauer gesagt aus Teilbereichen der Mechanik, muss Bestandteil der Engine sein. Dazu gehören auch Kollisionserkennung, Beschleunigung von Objekten und Verbindung von Objekten durch Gelenke.

Render Pipeline Eine Render Pipeline ist eine organisierte Abfolge von Berechnungen auf der Grafikeinheit eines Systems. Jede der Phasen in dieser Berechnungsabfolge ist unabhängig von der vorherigen. Dazu kommt, dass diese Schritte parallel berechnet werden können. [Gre14, S. 444 ff.] Die Anforderung an die Engine besteht in der Verfügbarkeit zur Ansteuerung der Renderpipeline.

Scripting In Game Engines soll die Möglichkeit bestehen, eine Skriptsprache festzulegen, welche es Entwicklern gestattet, selbst definierte Verhaltensweisen in eine Anwendung einzubringen. Solch eine Skriptsprache ermöglicht den Zugang zu Engine-Funktionen [Gre14, S. 794 ff.].

Inhalte nach Erstellung der Anwendung laden Bei der zu entwickelnden Software liegt ein großer Fokus auf dem dynamischen Laden von Inhalten. Damit ist gemeint, dass die Anwendung beispielsweise 3D-Modelle außerhalb der Entwicklungsumgebung der Engine laden kann. Dazu muss die Engine aber notwendige Funktionen zur Verfügung stellen.

3.2. Asuwahl der Engine

Diese Sektion widmet sich der Entscheidung, welche Engine innerhalb dieser Arbeit zum Einsatz kommt. Die Entscheidungsfindung gründet auf den gesammelten Informationen aus 2.2.1.

Alle betrachteten Engines verfügen über eine native Unterstützung der Virtual Reality Hardware HTC Vive. Desweiteren bieten alle native Unterstützung für Physik-Simulationen, welche den Ansprüchen dieser Arbeit genügen. Die Verwaltung der Renderaufgaben und die Kommunikation mit der Schnittstelle der Grafikeinheit ist ebenfalls eine von allen Engines unterstützte Fähigkeit.

Ein erster für die Entscheidung signifikanter Unterschied ist die Umsetzung des Skriptsystemes und damit einhergehende Programmiersprachen. Die Unity Engine unterstützt unter anderem C# und das .net Framework [Uni17,]. Vorteil hierbei ist, dass diese beispielsweise eine Bibliothek mit sich bringt, die Funktionen für das Auslesen von XML-Dokumenten liefert. Die Erfahrung des Autors mit der Unity Engine und der Programmiersprache C# ist ein weiterer Aspekt der für die Nutzung

dieser Sprache und die Unity Engine spricht. Die Cry Engine unterstützt unter anderem auch die Programmiersprache C#, deren Vorzüge bereits erwähnt wurden. Die Programmiersprache C++ wird von sowohl der Cry Engine als auch der Unreal Engine als Skriptmöglichkeit angeboten. Auch hier gibt es Bibliotheken, um XML-Dokumente zu bearbeiten [Cry17, Epi17,]. Die Erfahrung des Autors mit C++ sind sehr gering im Vergleich zu den C# Erfahrungen.

Die native Fähigkeit, Inhalte zur Laufzeit einer Anwendung zu laden, ist eine Funktion, die nur in der Unity Engine ohne eigenen Implementierungsaufwand möglich ist. Dieser Faktor fällt bei der Wahl der Engine stark ins Gewicht. Die in der Arbeit entstehende Anwendung soll das Laden von Assets durch Nutzer definierte Pfade unterstützen und ist damit auf eine solche Funktion angewiesen.

Aus diesen Gründen fällt die Wahl auch auf die Unity Engine. Die zum Einsatz kommende Programmiersprache ist C#.

3.3. Anforderung zur Datenhaltung

Es gilt festzulegen, welche Ansprüche an die Datenhaltung zu stellen sind. Die Anforderungen gehen aus den Zielen der Arbeit hervor, welche in 1.3 beschrieben sind. Diese müssen mit den Möglichkeiten der Unity Engine und der verwendeten Programmiersprache C# in Einklang gebracht werden. Im Abschnitt 3.6 ist beschrieben, dass alle konfigurierbaren Daten durch Zeichenketten repräsentiert sind. Außerdem ist die Möglichkeit der Erstellung von Aufzählungen ein Anforderung, da Elementtypen innerhalb einer Datei mehrfach auftreten können (siehe 3.6). Forderungen an das Datenhaltungsformat sind das strukturierte Halten von Zeichenketten und die Unterstützung von Aufzählungen eines Typs.

3.4. Auswahl des Datenhaltungsverfahrens

Bei der Wahl der Datenhaltung kommen keine Datenhaltungssysteme in Frage, die nicht über einer Auszeichnungssprache referenzierbar sind. Diese Entscheidung erfolgte auf folgender Basis. Datenhaltugssysteme, welche der Nutzer nicht selber

pflegen kann, sind mit einem hohen Implementierungsaufwand und fehlender Transparenz verbunden.

Vorteile des XML Formates

> „Erweiterbarkeit: XML erlaubt es, dem Benutzer eigene Tags oder Attribute zu definieren und zu benutzen, um seine Daten individuell zu strukturieren und mit Zusatzinformationen anzureichern.“ [RR03, S. 189].

Da Tags im XML-Format selbst definiert werden können, ergibt sich die Notwendigkeit, diese möglichst so zu benennen, dass diese dem Nutzer helfen, sich intuitiver in die angelegte XML-Struktur einzufinden. Durch die Vergabe von klaren eindeutigen Namen der Tags soll die Struktur an Übersicht gewinnen.

> „Struktur: XML ermöglicht die Darstellung beliebig tief verschachtelter Strukturen, wie man sie zur Repräsentation von komplexen Hierarchien aller Art benötigt.“ [RR03, S. 190].

Durch die hierarchische Baumstruktur sind XML-Dateien in ihren logischen inhaltlichen Aufbau gliederbar. Da ein XML-Knoten mehrere Elemente unter sich haben kann, ist die Möglichkeit Listen unter diesen Knoten zu erstellen gegeben.

> „Validierung: XML gibt externen Anwendungen die Möglichkeit, die strukturelle Gültigkeit der zu verarbeitenden Daten zu überprüfen. “ [RR03, S. 190].

Bei eventuell entstandenen strukturellen Fehlern besteht die Möglichkeit, eine Rückmeldung zu geben. Der erhoffte Effekt ist, dass beim Eintreten eines solchen Falles verkürzte Zeiten für das Korrigieren eines Fehlers anfallen.

Das XML-Format wird auf Grundlage der Vorteile und Erfüllung der Ansprüche als Datenhaltungsformat gewählt.

3.5. Auswahl der Entwurfsmuster

Dieser Abschnitt widmet sich den Vor- und Nachteilen ausgewählter Entwurfsmuster.

Singleton Das Singleton-Muster (siehe 2.4.2) dient dazu, maximal eine Instanz einer Klasse zu kreieren, sowie diese Klasse auch noch global erreichbar zu machen. Der Vorteil der globalen Erreichbarkeit ist beispielsweise, dass wenn viele Klassen Zugriff auf das Singleton-Objekt haben wollen, dieses über eine static-Variable der Singleton Klasse erreichbar ist.

Das Erzeugen maximal einer Instanz ist wichtig, wenn es kritische Klassen gibt, deren mehrfache Instanziierung Probleme verursachen kann. Für die Wahl der Nutzung sind beide Vorteile ausschlaggebend.

Factory Method Die Factory Method (siehe 2.4.2) übernimmt die Aufgabe der Instanziierung. Auch wenn dabei der Klassentyp der zu instanziierenden Klasse erst zur Laufzeit bekannt wird.

Unter der Nutzung des Strategy-Musters fallen viele Klassen an, welche von einer Überklasse erben. Vorteil ist, dass über die Factory Method diese verschiedenen konkreten Klassen gut instanziiert werden können. Auch wenn die konkrete Klasse erst zur Laufzeit bekannt wird.

Da das Factory Method-Muster gut zum Strategy-Muster passt, findet es Anwendung in dieser Arbeit.

Strategy Das Strategy Muster (siehe 2.4.2) gestattet es, Objekte mit verschiedenen Verhaltensweisen zu versehen, die zur Laufzeit veränderlich sind.

Da eine Funktionsweise des Toolkits darin besteht, AssetBundles und Verhaltensweisen für diese zu laden, ist das Strategy-Muster ein mit der Aufgabenstellung harmonisierender Lösungsanteil.

3.6. Architektur des Systems

Die zu entwickelnde Softwareplattform hat die Aufgabe, erweiterte Funktionalität in mit der Unity Engine entwickelter Programme einzubringen. Bei den zusätzlichen Möglichkeiten handelt es sich um das Laden von Assets und die Möglichkeit diese mit weiteren Funktionen zu versehen. Dieser Vorgang ist durch den Anwender definierbar. Für die Bedienung des Toolkits sollen keine weiteren Programmierkenntnisse

notwendig sein. Der Aufbau des Toolkits wird nachfolgend in diesem Abschnitt erörtert. Die Erklärungen bezieht sich auf C# als Programmiersprache.

Abstrakter Ablauf Es folgt die abstrakte Beschreibung der notwendigen Abläufe.

Der Anwender hat Zugriff auf eine Datensammlung. Diese Datensammlung beinhaltet Objekte und erweiterte Funktionen für selbige. Die Datensammlung ist nicht statisch, eine Erweiterung, bzw. Reduzierung des Datenbestandes ist möglich. In konfigurierbaren Dateien gibt er zu ladende Objekte und erweiterte Funktionen an. Zu einem definierten Punkt im Ablauf des Programmes lädt das System die zuvor angegeben Objekte und ihre erweiterte Funktionen soweit sie angegeben wurden.

Umsetzung zu konkretem Ablauf Auf den eben beschriebenen abstrakten Ablauf basierend, folgt der konkrete Ablauf.

Bei den vom Nutzer referenzierbaren Objekten des Ablaufes handelt es sich um Assets (siehe 2.2.2), da diese die nutzbaren Objekte innerhalb der verwendeten Unity Engine sind. Das Laden von Assets zur Laufzeit ist nativ nur durch AssetBundles möglich (siehe 2.2.2). AssetBundles werden über den Pfad im Dateisystem angegeben (siehe 2.2.2). Das konfigurierbare Datenhaltungsformat muss folglich diesen Pfad beinhalten.

Die erweiterte Funktionalität für diese Objekte besteht aus einer Verknüpfung der Objekte mit Klassen im Sinne der objektorientierten Programmierung (siehe 2.4.1). Klassen, die die Funktionalität innerhalb von Unity darstellen, befinden sich innerhalb eines Skripts (siehe 2.2.2). Laut [Mic17,] kann eine Klasse anhand ihres Namens instanziiert werden. Das Datenhaltungsformat muss diesen Namen beinhalten können, um den Ladevorgang überhaupt zu ermöglichen.

Eine Klasse kann zu ihrer Instanziierung Werte übergeben bekommen [Mic17,]. Funktionen sind dadurch konfigurierbar. Das Datenhaltungsformat muss solche Werte beinhalten können. Dafür ist die Angabe von Datentypen essenziell. Grund dafür ist, dass Inhalte eines XML-Dokumentes Zeichenketten sind (siehe 2.3), aber auch andere Datentypen instanziierbar seien sollen. Ganzzahlen (int [Mic17,]), Gleitkommazahlen (float [Mic17,]), Wahrheitswerte (bool [Mic17,]) und Zeichenketten (string [Mic17,]) sind Datentypen, die präventiv eingebunden werden.

Das Datenhaltungsformat XML besitzt die Fähigkeiten, um den genannten Anforderungen zu genügen (siehe 3.4). Konfigurationen des Nutzers finden deshalb innerhalb von XML Dateien statt. AssetBundles, Klassen und Werte werden in XML-Daten definiert. Um den Überblick in den XML-Dateien zu wahren und um bereits definierte Objekte wiederverwenden zu können, findet die Datenhaltung in zwei verschiedenen XML-Dokumenttypen statt. Einer ist der als XML-Objekt bezeichnete Dokumenttyp. Dieser enthält Informationen zu einem Objekt, bestehend aus maximal einem AssetBundle, möglicher Klassen sowie weitere den Klassen zugehörige Werte. Der andere Dokumenttyp, XML-Startbedingung genannt, listet die XML-Objekte, deren Referenzen beim Ausführen der Softwareplattform geladen werden. Dazu ist das Einbinden des Toolkits und das definieren eines Ladezeitpunktes notwendig.

Ordnerstruktur Zusätzlich bietet sich die Erstellung eines Ordnersystemes an, um die Datenhaltung übersichtlicher gestalten zu können. Die Erwartung ist, dass Nutzer einen besseren Einstieg in die Arbeit mit dem Toolkit bekommen und durch geordnete Strukturen effizienter arbeiten können. Ein eigenes Ordnersystems sorgt unterdessen auch dafür, dass keine Vermengung mit dem Toolkit fremder Dateien stattfindet. Ein nicht statisches Ordnersystem ist gefragt, um die Mobilität der Daten zu wahren. Der Pfad zu einem Einstiegspunkt in die Ordnerstruktur wird dem Toolkit mitgeteilt. Das ist nützlich, wenn mehrere Wurzelpfade für unterschiedliche Projekte nutztbar sein sollen.

3.7. Analyse der Usability-Verfahren

Usability ist ein wichtiges Thema. Wie in 2.5 beschrieben, gilt es, dem Anwender durch geeignete Verfahren die Funktionsweise und Bedienung der Anwendung durch Feedbacks näher zu bringen.

Akustisches Feedback Die verwendete VR-Brille HTC Vive besitzt einen Audioausgang und bietet damit die Möglichkeit, akustische Signale zu verwenden. Die Anwendung, die im Rahmen dieser Arbeit entsteht, verzichtet aber auf den Gebrauch des Audioausgangs und damit auch auf akustisches Feedback.

Visuelles Feedback Einige Spiele bedienen sich der Farbe Grün als Indikator für Positionierungen. In diesen Fällen handelt es sich zumeist um ein Objekt, welches an einer bestimmten Position angebracht werden soll. Befindet sich das Objekt an einer zulässigen Stelle, so färbt es sich grün und indiziert damit eine legitime Handlung. Beispiele für Umsetzungen dieser Praktik in 3D-Anwendungen sind in der Abbildungen 3.1 zu sehen. Das gleiche verfahren findet auch in der Pipettieranwendung statt. Beim Zurückstellen einer Pipette in den Pipettenhalter färbt sich die Pipette grün und signalisiert dem Anwender, dass er die Pipette loslassen kann. Ähnliches passiert bei den Eppendorfgefäßen. Da diese sehr klein sind, ist es schwierig, sie zurück in den Gefäßbehälter zu legen. Zur Vereinfachung werden diese in dem Gefäßbehälter abgelegt, die Einsortierung geschieht automatisch. Die Gefäße färben sich grün, wenn sie sich an der richtigen Position zum Ablegen befinden.

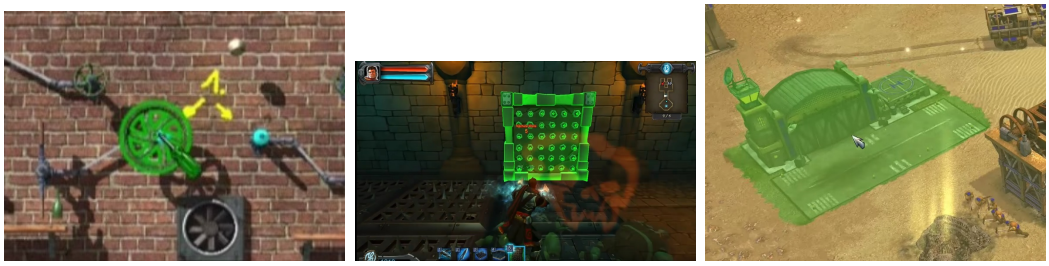


Abbildung 3.1.: Links: Ein Screenshot aus dem Spiel Crazy Machines 2 von Fakt Software. Zu sehen ist, dass das zu positionierende Rad grün gefärbt ist, um den Spieler zu suggerieren, dass die Anbringung dort möglich sei. Mitte: Ein Screenshot aus dem Spiel Orcs Must Die! von Robot Entertainment. Zu sehen ist, dass der Spieler gerade eine Falle an der Wand anbringen möchte. Diese ist als Feedback grün eingefärbt und kann an der gewünschten Position angebracht werden. Rechts: Ein Screenshot aus dem Spiel Empire Earth 2 von Mad Doc Software. Der Spieler sucht sich einen Bauplatz für ein Gebäude aus, dieses ist grün gefärbt da der aktuelle Ort ein gültiger Bauplatz ist.

Haptisches Feedback Die Controller der verwendeten VR-Brille HTC Vive bieten die Möglichkeit für haptisches Feedback. Haptisches Feedback wird wie folgt umgesetzt. Die verwendeten Pipetten verfügen über 2 Druckpunkte, wie in 2.6 beschrieben. Wenn eine Druckstufe erreicht ist, so wird eine Vibration im Controller ausgelöst und der Nutzer wird über das Erreichen der Druckstufe informiert. Die

Umsetzung des Feedbacks durch die haptischen Fähigkeiten des Controllers, anstelle einer visuellen Umsetzung, sorgen zu einem gewissen Teil dafür, dass die visuellen Feedbacks nicht unübersichtlich werden.

Feedback Fehler Toolkit Das dynamische Laden von Komponenten durch das Toolkit erfolgt durch Pfadangaben innerhalb von XML Dateien. Tippfehler in den Angaben sorgen dafür, dass die Anwendung nicht mehr korrekt ausgeführt wird. Ein Fehlerfenster, welches die fehlerhafte Datei und die Art des Fehlers anzeigt, sorgt dafür, diese Fehler schneller zu finden und beseitigen zu können.

4. Implementierung

Im Anschluss an die Anforderungen und die Grundlagen findet in diesem Kapitel die Ausführungen zur Implementation statt. Auf die Struktur der Software, sowie auf essentielle bzw. besondere Bestandteile wird eingegangen. Alle Begriffe aus der Programmierung sowie Quellcode-Beispiele sind auf die Programmiersprache C# bezogen. Zu beginn erfolgt die Definition einer Ordnerstruktur, auf welchen die folgenden Punkte basieren. Darauf aufbauend findet die Betrachtung der XML-Dokumente statt, im Anschluss stehen die Objekte der Programmierung im Fokus, die aus den XML Dokumenten entstehen, bevor auf Funktionen und Möglichkeiten zur Erweiterbarkeit des Toolkits eingegangen wird.

4.1. Ordnerstruktur

Die Erstellung einer Ordnerstruktur erfolgt aufgrund der Anforderung im Abschnitt 3.6. Dabei existieren drei zu ordnende Kategorien: die XML-Objekt-Dateien, die XML-Startbedingungen und die AssetBundles. Für jede dieser Kategorien existiert ein eigener Ordner. Diese Ordner wiederum sind in einem übergeordneten Ordner enthalten und können selber Unterordner beinhalten. Daraus ergibt sich für das Toolkit eine Ordnerstruktur, welche in Abbildung 4.1 dargestellt ist. Die oberste Ebene dieser Struktur bildet ein Ordner, welcher Wurzelverzeichniss genannt wird und den Einstiegspunkt für das Toolkit liefert. Zusätzlicher Effekt ist, dass sich Pfadangaben für XML-Dateien dadurch verkürzen, denn XML-Dateien sowie Assetbundles liegen innerhalb desselben Wurzelverzeichnisses vor. Um die kürzeste Schreibweise zu erhalten, beginnen die Pfade für Referenzierungen von Dateien jeweils von ihrem zuständigen Ordner (siehe 4.6.4).

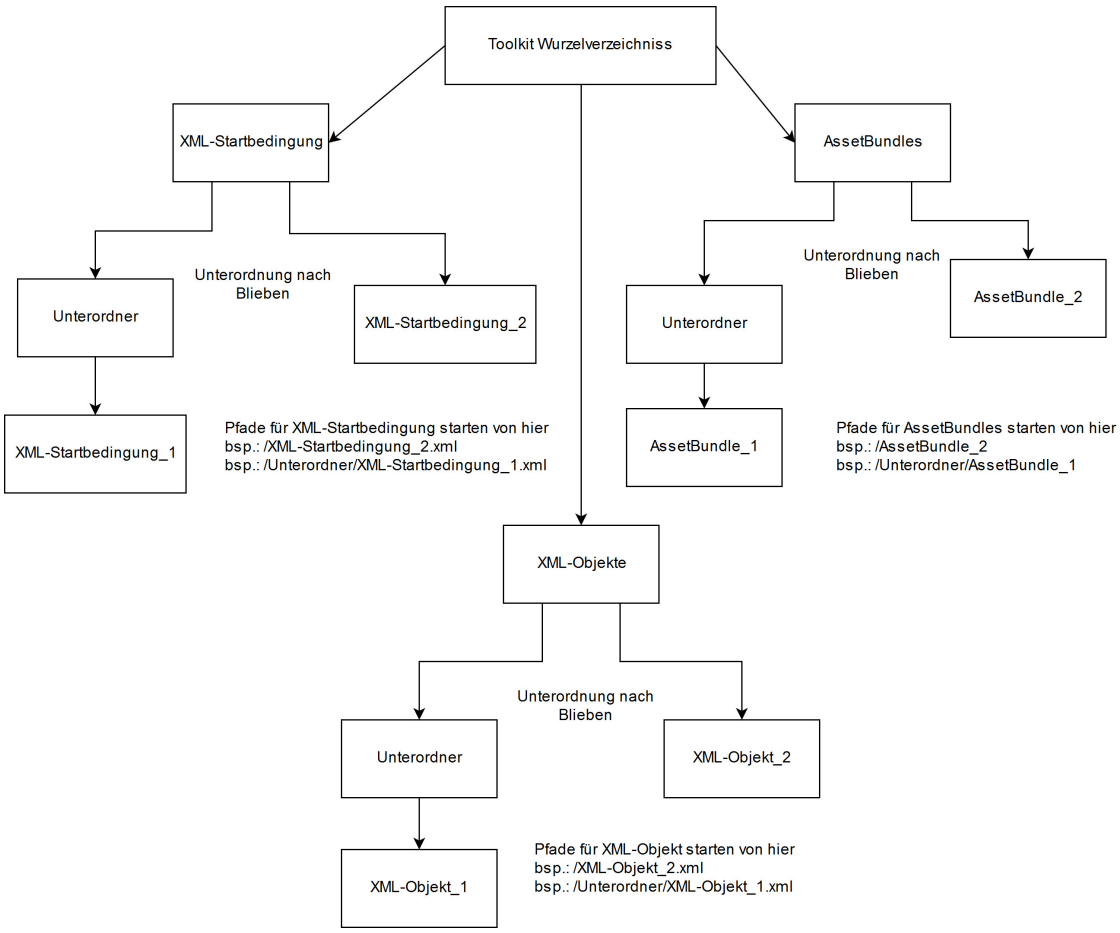


Abbildung 4.1.: Toolkit Ordnerstruktur

4.2. Verwendung von XML

Anhand der Festlegungen im Abschnitt 3.6 gelangt XML als Format für die Referenzierung von Daten zum Einsatz. Zwei Arten von XML-Dokumenten sind zu realisieren. Eine für die Informationen über Assetbundles und zugehörige Verhaltensweisen (XML-Objekt) und eine andere für die Gesamtheit der zu ladenden XML-Objekte (XML-Startbedingung). Ausführungen der beiden hierarchisch strukturierten XML-Dokumenttypen finden im Folgenden statt.

Grundgerüst XML-Dateien Um die Prüfung verwendeter XML-Dateien auf ihre Art (XML-Objekt oder XML-Startbedingung) zu ermöglichen, verwenden alle für das Toolkit gültige XML-Dateien die gleichen Tags in der oberen Hierarchieebene.

Oberstes Element ist immer das <Dokument>-Tag. Es ist der Einstieg in das Dokument. Eine Ebene tiefer befindet sich das <Type>-Tag, es wird genutzt, um den Dokumenttyp festzustellen. In dem <Type>-Tag kann eine Zeichenkette definiert werden. Nur wenn sich darin eine gültige Zeichenkette befindet, kann eine Überprüfung erfolgen. Die gültigen Zeichenketten 'objekt' (für XML-Objekt) oder 'szenario' (für XML-Startbedingung). Die Typenprüfungen finden immer beim Einlesen einer XML-Datei statt. (siehe 4.6.3 , 4.6.2).

C# XML-Funktionen Im .net Framework befinden sich unter anderem Funktionen für die Arbeit mit XML-Dateien (siehe 3.2). In dieser Arbeit ist es der System.XML namespace (engl. Namensraum), welcher zum Einsatz kommt. Bereitgestellte Klassen sind beispielsweise XmlDocument, XmlNodeList und XmlNode. Diese stehen für ein XML-Dokument (XmlDocument), einen XML-Knoten (XmlNode) und eine Sammlung von XML-Knoten (XmlNodeList). Zusätzlich zu den Klassen liefert der namespace auch Funktionen. Aus dem Dokument ist es möglich, durch SelectNodes eine XmlNodeList zu erhalten, SelectSingleNode extrahiert eine XmlNode druch ihren Namen. Eine XmlNodeListe kann einzelne XmlNodes zurückgeben. Aus einer XmlNode können Unterknoten oder ein Wert ausgelesen werden. [Mic17,]

4.2.1. XML-Objekte

Überblick XML-Objekt Die XML-Objekte sind Kernelemente des Toolkits. Sie beinhalten alle Informationen, um AssetBundles und Verhaltensweisen in Form von Behaviour-Klassen (siehe 4.4) zu einem nutzbaren Objekt zusammenzuführen. Das XML-Objekt ist als ein Baukasten im XML-Format zu verstehen und damit für die Konfiguration auf Objektebene (siehe 4.3) zuständig. Die anzugebenden Werte innerhalb einer Objekt-XML sind ein Pfad zu einem AssetBundle, mehrere Behaviour-Klassen und mehrere Werte für Behaviour-Klassen. Keine dieser Angaben ist eine Pflicht. Es gibt eine Abhängigkeit in diesem Zusammenhang. Werte für Behaviour-Klassen können nur in Behaviour-Klassen angegeben werden.

Aufbau des XML-Objekts Dementsprechend angelegte Tags dienen zur Angabe der Informationen. In der Abbildung 4.2 sind die XML-Hierarchie des XML-

Objektes, sowie alle definierten Tags zu sehen. Im weiteren Fortgang erfolgt eine Erklärung der abgebildeten XML-Elemente.

Objekt-Tag Das <Objekt>-Tag ist der Knoten, unter welchem sich alle Konfigurierbaren Unterknoten befinden.

ModellPfad-Tag Der <ModellPfad>-Tag ist die Referenz zu einem Assetbundle. Ist es nicht angegeben, so wird kein AssetBundle geladen. Die Angabe von AssetBundles ist keine Voraussetzung für die Funktionalität eines XML-Objektes. Das Element enthält keine weiteren Unterknoten. Der Pfad wird als Zeichenkette eingegeben.

BehaviourListe-Tag <BehaviourListe> ist der Tag, unter welchem Behaviour-Elemente anzugeben sind. Dieser Knoten existiert, damit die strukturierte Angabe mehrerer Behaviours möglich ist.

Behaviour-Tag Das <Behaviour>-Tag repräsentiert eine Behaviour-Klasse und enthält in seinen Unterknoten alle Informationen- um eine solche zu instanziieren.

Name-Tag Innerhalb des <Name>-Tags befindet sich der Name der zu instanziierenden Klasse. In diesem Element sind keine weiteren Knoten anzugeben. Der Name wird als Zeichenkette angegeben.

WerteListe-Tag <WerteListe> ist der Tag, unter welchem Werte Elemente anzugeben sind. Dieser Knoten existiert, damit die strukturierte Angabe mehrerer Werte möglich ist.

WertObjekt-Tag Ein <WertObjekt> repräsentiert einen konkreten Wert innerhalb des <WerteListe> Tags. Er besteht aus Unterknoten, welche den Wert definieren.

```
<?xml version ="1.0" encoding="UTF-8"?>
<!-- Das Objekt_Template zeigt wie eine XML-Objekt Datei aufgebaut ist -->
<Document>                    <!-- Hier beginnt das Dokument  -->
    <Type>Objekt</Type>
    <Objekt>
        <Beschreibung>Optionale beschreibung des Objektes</Beschreibung>
        <ModellPfad>\Beispielpfad\AssetBundle-Datei</ModellPfad> <!-- Pfad zu einem AssetBundle  -->
        <BehaviourListe>        <!-- Knoten unter dem verschiedene Behaviours definiert werden  -->
            <Behaviour>         <!-- Ein Behaviour Element -->
                <Name></Name>   <!-- Name des Behaviour Elements -->

                <WerteListe>    <!-- Knoten unter dem verschiedene für Werte des Bahviours definiert werden -->
                    <WertObjekt>    <!-- Ein Wert Element -->
                        <WertTyp></WertTyp> <!-- Typ des Wertes -->
                        <Wert></Wert>       <!-- Der tatsächliche Wert -->
                    </WertObjekt>
                </WerteListe>
            </Behaviour>
        </BehaviourListe>
    </Objekt>
</Document>
```

Abbildung 4.2.: Das XML-Template für das XML-Objekt

WertTyp-Tag Der <WertTyp>-Tag gibt den Datentyp des zu instanziierenden Wertes an. Die Notwendigkeit hierfür ist in dem Abschnitt 3.6 erläutert. Namen der Datentypen (siehe 3.6) agieren als Identifikationsmerkmal. In diesem Element sind keine weiteren Knoten anzugeben, der Name des Datentyps wird als Zeichenkette angegeben.

Wert-Tag Im <Wert>-Tag ist der tatsächliche Wert enthalten. Um die Funktionalität zu gewährleisten, ist es ratsam, dass der eingetragene Wert sich an die Konvention seines zugehörigen Datentyps hält [Mic17,] . Der Wert wird seiner Konvention entsprechend als Zeichenkette eingetragen.

4.2.2. XML-Startbedingung

Die XML-Startbedingung legt die Ausgangssituation des Anwendungsszenarios fest. Alle durch das Toolkit zu ladenden AssetBundles und Behaviour-Klassen finden sich in der XML-Startbedingung wieder. Nicht direkt, sondern über die zugewiesenen XML-Objekte. Die XML-Startbendingung hat die Aufgabe durch Verweise von Pfadangaben auf XML-Objekte indirekt dafür zu sorgen, dass diese beim Programmstart geladen, ausgelesen und deren Inhalt geladen wird.

Aufbau der XML-Startbedingung In Abbildung 4.3 ist die Struktur der XML-Startbedingung zu sehen. Der Aufbau des Dokuments und die Bedeutung der Tags findet im folgenden statt.

Inhalt-Tag Der <Inhalt>-Tag ist der Knoten, unter welchem sich alle konfigurierbaren Unterknoten einer XML-Startbedingung befinden.

Umgebung-Tag Innerhalb des <Umgebung>-Tags findet die Angabe eines AssetBundles durch eine Pfadangabe statt. Dieses fungiert als Kulisse. Die Angabe ist keine Pflicht. Der Pfad wird als Zeichenkette angegeben.

StartObjekte-Tag <StartObjekte>-Tag ist eine Liste, dessen Unterknoten aus Objekt Elementen besteht.

Objekt-Tag Das <Objekt>-Tag steht für ein einzelnes XML-Objekt Element. Alle nötigen Daten, um das beschriebene XML-Objekt zur Instanziierung weiterzuleiten befinden sich in seinen Unterknoten.

Pfad-Tag Der Inhalt des <Pfad>-Tags verweist auf eine XML-Objekt-Datei, welche das Toolkit bei Ausführung lädt. Die Pfadangabe findet als Zeichenkette statt.

```
<Document>

    <Type>Szenario</Type>

    <Inhalt>

        <Beschreibung>hier ein Beschreibungstext für das Szenario einfügen</Beschreibung>

        <Umgebung>hier sollte der Pfad zu einem AssetBundle stehen, welches als Umgebung Fungiert</Umgebung>

        <StartObjekte>

            <Objekt>

                <Pfad>Hier einen Pfad zu einem XML-Objekt einfügen, Ordnestruktur beachten</Pfad>

                <Position>

                    <xPosition>Hier Die X-Koordinate einfügen</xPosition>
                    <yPosition>Hier Die Y-Koordinate einfügen</yPosition>
                    <zPosition>Hier Die Z-Koordinate einfügen</zPosition>

                </Position>

            </Objekt>
        </StartObjekte>
    </Inhalt>
</Document>
```

Abbildung 4.3.: Das XML-Template für die XML-Startbedingung ohne relevante Header-Informationen

Position-Tag Bei der Angabe eines AssetBundles mit 3D-Darstellungen im Pfad Element ist das <Position>-Tag nützlich, insofern es eine bestimmte Position einnehmen soll. Die <X>-, <Y>- und <Z>-Tags sind Unterknoten des Position-Tags. In ihnen findet die Angabe einer Position als Zeichenkette auf der jeweiligen Achse statt. Die Angabe der Position ist nicht zwingend erforderlich.

4.3. Objekt-Klassen

Als Resultat des Strategie-Musters (siehe 2.4.2) übernehmen Behaviour-Klassen die Funktionalität (siehe 4.4). Eine Verwaltung sowie die Schnittstelle zu einem GameObject sind notwendig. Die Objekt-Klasse kommt diesen Aufgaben nach. Die Objekt-Klasse ist das Resultat eines XML-Objektes. Jedes geladene XML-Objekt erzeugt genau eine Objekt Klasse. Behaviour-Klassen und AssetBundles aus einem XML-Objekt werden mit einer zugehörigen Objekt-Klasse verknüpft.

Funktionsweise Objekt-Klasse In der Abbildung 4.4 ist die UML-Darstellung des Objekt-Klasse zu sehen. Die Objektklasse erbt von der MonoBehaviour-Klasse (siehe 2.2.2). Dadurch ist es der Objekt-Klasse möglich die Update-Funktion und Funktionen, die sich auf das zugehörige GameObjekt beziehen zugreifbar zu machen [Uni17,]. Eine Liste vom Typ Behaviour befindet sich in der Objekt-Klasse. Diese sammelt die zugewiesenen Behaviour-Klassen und verwaltet sie. AddBehaviour- und RemoveBehaviour-Funktionen sind für die Referenzierung beziehungsweise der Dereferenzierung von Behaviour-Klassen zuständig. Behaviour-Klassen besitzen eine Methode namens updateBehaviour (siehe 4.4). Der Aufruf erfolgt in der Update-Funktion der Objekt-Klasse. Eine konkrete Instanz der Objekt-Klasse aktualisiert damit alle mit ihr referenzierten konkreten Behaviour-Klassen regelmäßig über diese Methode.

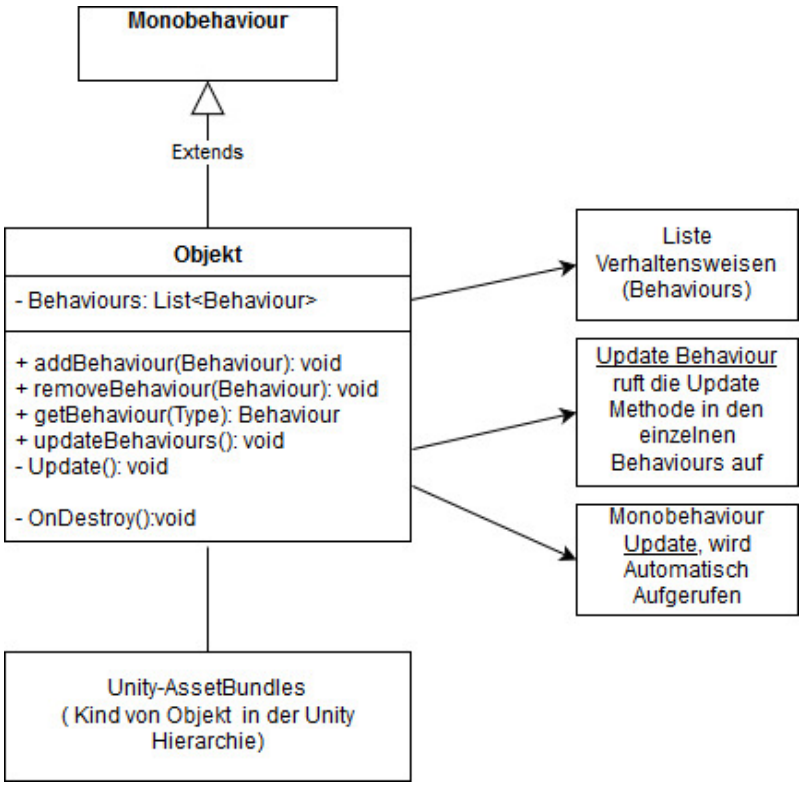


Abbildung 4.4.: Die Objekt-Klasse in ihrer UML-Darstellung mit Kommentaren

4.4. Behaviour-Klassen

Behaviour-Klassen übernehmen die Funktionalität innerhalb des modularen Aufbaus. In ihnen befinden sich Anweisungen für das Verhalten der AssetBundles, welche als GameObjects geladen werden. Sie sind eine Implementierung des Strategie, Entwurfsmusters (siehe 2.4.2). Für die Umsetzung des Strategie-Musters ist es notwendig, dass alle konkreten Verhaltensweisen ein Interface implementieren. Dieses Interface sorgt dafür, dass die konkreten Verhaltensweisen durch das Interface referenzierbar sind und Objekte die Möglichkeit haben, über Schnittstellen auf Funktionen der konkreten Verhaltensweisen zuzugreifen. Das zur Behaviour-Klasse zugehörige Klassendiagramm 4.5 dient zur Orientierung.

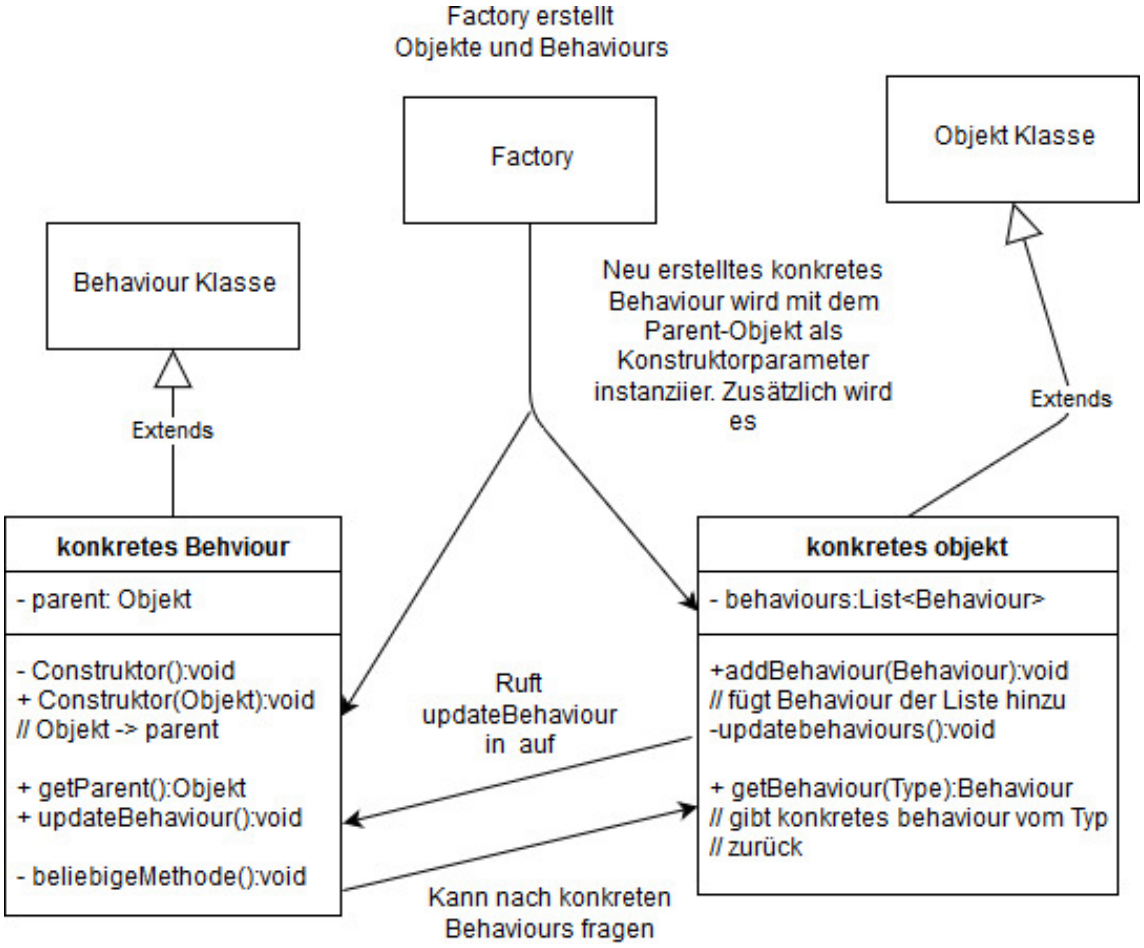


Abbildung 4.5.: Relationen der Behaviour- und Objekt-Klasse in UML-Darstellung.

Abstrakt statt Interface Das Strategy-Muster empfiehlt die Verhaltensweisen durch ein Interface zu abstrahieren. Aus folgenden Gründen findet aber eine Abstrahierung durch eine abstrakte Klasse statt. Für die Umsetzung ist es notwendig, die Verhaltensweisen mit mehreren Informationen zu instanziieren. Die Verhaltensweisen sollen durch XML-Dateien konfigurierbar sein (siehe 3.6). Da das Fatory-Muster die Verhaltensweisen instanziiert, ist es erforderlich, dass der Konstruktor der konkreten Klassen nicht statisch ist. Die Aufgaben des Interfaces, deren Zugriff auf Funktionen und übergeordnete Referenzierbarkeit sind auch durch die abstrakte Klasse möglich.

Aufbau des Konstruktors Die Werte aus einer XML-Objekt Datei werden dem Konstruktor übergeben. Die Hauptaufgabe der Verhaltensweisen ist die Manipulation von GameObjects. Diese sind über die Objekt-Klasse erreichbar. Daher stellt die Behaviour-Klasse Funktionen für den Zugriff auf die zugehörige Instanz der Objekt-Klasse bereit. Es wichtig, dass die Behaviour-Klasse eine Referenz zu dieser Objekt-Klasse besitzt. Es gilt also durch den Konstruktor sicherzustellen, das Werte übergeben werden können und eine Objekt-Klasse existiert.

Objekt-Referenz durch Konstruktor herstellen Die Behaviour-Klasse, von der alle Verhaltensweisen erben, besitzt einen Konstruktor, der eine Referenz zu einer Objekt-Klasse übergeben bekommt. Diese Referenz wird gespeichert und durch die Funktion getBehaviour für das konkrete Behavior zugänglich gemacht. Damit ein Behaviour nur mir einer Referenz zu einer Objekt-Klasse erstellt werden kann, ist der Standard-Konstruktor auf die private Sichtbarkeit gestellt. Der Effekt ist, dass die konkrete Behaviour-Klasse nicht über den Standard-Konstruktor, folglich nicht ohne Objekt-Klassen-Referenz, instanziiert werden kann. Da die konkreten Behaviour-Klassen unterschiedliche Konstruktoren verwenden, ist es möglich, unterschiedliche Konstruktoren für konkrete Behaviours sowie die Behaviour-Klasse zu implementieren. Die konkreten Behaviour-Klassen können in ihrem Konstruktor die Referenz zur Objekt-Klasse an den Konstruktor der Behaviour-Klasse weiterleiten. Dies ist mit dem Base-Befehl der Programmiersprache C# möglich [Mic17,].

Werte durch den Konstruktor übergeben Das Instanziieren einer Klasse durch Referenz zu ihren Namen ist mit einer Verknüpfung von zwei Funktionen der Programmiersprache C# möglich. Dazu ist zuerst der Klassentyp zu spezifizieren. Dies

geschieht durch die Funktion Type.GetType, welche als Parameter den Namen der zu erstellenden Funktion als Parameter in Form einer Zeichenkette übergeben bekommt [Mic17,]. Als nächstes erfolgt die Instanziierung des Klassentypes durch die Activator.CreateInstance-Methode, welche über den Konstruktor beispielsweise den Klassentype und Werte übergeben bekommt (siehe 4.6.3) [Mic17,].

Aktualisieren der konkreten Behaviour-Klassen während der Laufzeit Behaviours, deren Verhaltensweisen auf kontinuierlichen Aktionen basieren, benötigen eine Institution, welche für die wiederkehrenden Aufrufe von Funktionen in den Behaviours zuständig ist. In der MonoBehaviour-Klasse ist eine solche Funktion namens Update implementiert (siehe 2.2.2). Die Objekt-Klasse erbt von der MonoBehaviour-Klasse. In ihr ist diese Funktion nutzbar. Damit die Objekt-Klasse seine zugehörigen Behaviour-Klassen aktualisieren kann, ist in der Behaviour-Klasse die abstrakte Methode behaviourUpdate Definiert. Die Objekt-Klasse traversiert in ihrer Update-Funktion alle ihr bekannten konkreten Behaviour-Klassen und ruft in ihnen die behaviourUpdate-Methode auf. Die konkreten Behaviour-Klassen implementieren, der Aufgabe ihrer Verhaltensweisen entsprechend, Programmcode in dieser Funktion. Es ist aber auch möglich, diese leer zu lassen.

Behaviour-Katalog Ein Katalog ist im Wurzelverzeichniss angelegt. Dieser enthält einen Überblick über die vorhandenen Behaviour-Klassen. Namen der Behaviour-Klassen, Aufbau der Konstruktoren, Abhängigkeiten und Beschreibungen der jeweiligen Funktion sind in diesem Katalog enthalten. Damit soll dem Nutzer eine Möglichkeit eines Nachschlagewerkes gegeben werden. Damit dieses konsistent bleibt, sind alle Neuer- und Änderungen an Behaviour-Klassen in diesem Katalog zu vermerken.

4.5. Erweiterbarkeit

Die Nutzung des Toolkits findet im Rahmen dieser Arbeit mit dem konkreten Anwendungsfall des Pipettierens statt. Die Anforderung an den Aufbau ist aber nicht die Funktionalität für einen einzelnen statischen Anwendungsfall. Das Toolkit soll für individuelle dynamische Nutzung im Bereich von modularen Szenario-Aufbau geeignet sein. Zusätzlich soll das Toolkit offen für Erweiterungen sein.

AssetBundles-Erweiterung Die AssetBundles befinden sich in einem zugänglichen Ordner der Toolkit Ordnersturktur. Dort ist es möglich, diese nach belieben zu löschen, verschieben, umzubennen und neue hinzuzufügen. Neue Assetbundles müssen lediglich in den AssetBundle-Ordner verschoben werden. Die Angabe ihres Pfades in einer XML-Objekt-Datei, welche wiederum in der genutzten XML-Startbedingung vorkommt, ist ausreichend, um diese zu laden. Ähnlich verhält es sich mit Änderungen oder Neuerungen der XML-Dateien.

XML-Dateien-Erweiterung Damit XML-Dateien für Änderungen und Erweiterungen offen sind befinden sich, diese in einem zugänglichen Ordner innerhalb der Toolkit Ordnerstruktur. Um XML-Dateien zu ändern oder zu erzeugen, genügt die Editierung. Beziehungsweise Erstellung, mithilfe eines Texteditors, welcher die UTF-8-Kodierung Unterstützung. Bei der Erstellen von neuen XML-Dateien findet der Nutzer Unterstützung durch Templates (dt. Vorlagen) zu den XML-Dateitypen. Bei Änder- und Neuerungen ist es essenziell, dass die Ordnerstruktur des Systems eingehalten wird, da hierbei andernfalls Fehler auftreten, welche das Toolkit an der Ausführung hindern. Um beispielsweise eine neue XML-Starbedingung zu verwenden, ist es erforderlich, dem Softwaresystem ihren Pfad anzugeben (siehe 4.6.4). Ähnlich verhält es sich mit einem neuen XML-Objekt. Um dieses zu nutzen muss sein relativer Pfad vom Objekt-XML-Ordner ausgehend in einer XML-Startbedingung auftauchen. Zusammenfassend verhält es sich mit neuen Assetbundles, XML-Startbedingungen und XML-Objekten so, dass diese in ihre jeweiligen Ordner verschoben und durch Referenzpfade im späteren Progammablauf geladen werden. Desweiteren ist es realisierbar XML-Dateien in ihrer Hierarchie unter der Bedingung zu erweitern, dass die aktuellen Relationen bestehen bleiben.

Behaviour-Erweiterung Bei Behaviour-Klassen ist die Lage hingegen anders. Sie befinden sich nicht in Ordnern außerhalb des Programms. Die Instanziierung ist zur Laufzeit zwar möglich, aber nicht die Definition. Bei Änder- und Neuerungen ist es unausweichlich, das Programm neu zu übersetzen. Nur so ist es möglich, bestehende Klassen von der Existenz der neu hinzugekommenen zu informieren. Dieser Vorgang verlangt das Unityprojekt (siehe 2.2.2) innerhalb der Unity Engine zu öffnen. Dort besteht die Möglichkeit, Klassen innerhalb von Skripten zu bearbeiten. Das Anlegen neuer Skripte ist ebenfalls durchführbar. Um neue konkrete Behaviour-Klassen anzulegen erben diese von der abstrakten Behaviour-Klasse. Zusätzlich ist

es erforderlich, die abstrakten Funktionen der Behaviour-Klasse zu implementieren. Anschließend ist es unerlässlich, das Projekt zu speichern und ein neues Build (siehe 2.2.2) zu erstellen. Dabei einsteht das komplette Programm neu und enthält alle Behaviour-Änderungen.

4.6. Funktionen des Systems

Diese Abschnitt befasst sich mit der Implementierung des Toolkits, welches die Aufgaben des Ladens und Instanziieren übernimmt.

4.6.1. Singleton mit MonoBehaviour-Klassen

Das Singleton-Muster unter Verwendung von MonoBehaviour-Instanzen (siehe 2.2.2) gestaltet sich anders als im Muster beschrieben. Die Erzeugung von MonoBehaviour-Klassen durch Skripts wird durch die addComponent Funktion realisiert. Diese übernimmt die direkte Instanziierung, welche im Singleton-Muster unterbunden wird (siehe 2.4.2). Um trotz der Instanziierung durch MonoBehaviour ein Singleton zu realisieren, muss nach der Instanziierung eine Prüfung stattfinden, ob bereits eine Instanz der Klasse existiert. Falls nicht, referenziert sich die Instanz selber in ihrer static-Variable vom Typ der eigenen Klasse. Andernfalls zerstört sich die Instanz umgehend selbst durch die MonoBehaviour Funktion Destroy (siehe [Uni17,]). In der Abbildung 4.6 ist ein MonoBehaviour Singleton Umgesetzt.

4.6.2. Startbedingungen laden

Die Datenhaltung findet auf zwei Ebenen statt. Zu einem gibt es die XML-Startbedingung, welche ein ganzes Szenario definiert und die XML-Objekte, welche einzelne Objekte darstellen und als Bausteine für diese Szenarios fungieren. Die *StarbedingungAuslesen-Klasse* übernimmt die Aufgabe die XML-Startbedingung zu laden und Auszulesen. Anschließend leitet sie die XML-Objekte die sich in einer XML-Startbedingung befinden an die Facory weiter, wo ihre Instanziierung erfolgt.

```
public class Factory : MonoBehaviour {


    public static Factory singelton;

    void Awake(){
        if (singelton == null) {
            singelton = this;
        } else {
            Destroy (this.gameObject);
        }
    }
```

Abbildung 4.6.: Die Umsetzung eines Singletons innerhalb einer MonoBehaviour-Klasse

Die *StartbedingungKlasse* implementiert eine Funktion namens *StartbedingungAusführen*. Sie erhält beim Aufruf über ihren Konstruktor einen Pfad zu einer XML-Startbedingung in Form einer Zeichenkette. Die Funktion lädt das XML-Dokument als Zeichenkette und parst es anschließend in ein XmlDocument (siehe 2.3). Anschließend prüft sie auf den Inhalt des <Type>-Tags und sucht diee Elemente vom Tag-<Objekt> innerhalb des <ObjekteListe>-Knotens. Jedes Element vom Typ <Objekt> wird an die Factory weitergeleitet. Diese Übernimmt anschließend die Aufgabe der Instanziierung.

4.6.3. Die Factory-Klasse

Die Factory-Klasse übernimmt vorrangig folgende Aufgaben, in ihr findet das Auslesen der XML-Objekte und deren Instanziierung dieser durch die Umsetzung des Entwurfsmusters Factory-Method statt. In ihr findet zudem die Implementierung des Singleton-Musters Anwendung.

XML-Objekte auslesen In 4.6.2 ist beschrieben wie XML-Startbedingungen ausgelesen und verarbeitet werden. Anschließend findet die Weiterleitung der ausgelesenen XML-Objekte an die Factory statt, wo ihre Instanziierung erfolgt. Die Factory-Klasse erhält eine Funktion namens InstanziiereObjekt. Diese bekommt einen Xml-Node im Konstruktor übergeben und erwartet, dass es sich um ein <Objekt>-

Element aus einer XML-Startbedindung (siehe 4.2.2) handelt. Dieser Knoten enthält den Pfad zur eigentlichen XML-Objekt-Datei innerhalb des <Pfad>-Tags. Der Text der XML-Objekt-Datei wird geladen und zu einem XmlDocument geparst. Dort findet zuerst die Abfrage des <Type>-Tags (siehe 4.2) statt. Ergeben sich keine Probleme folgt das Laden der AssetBundles und Behaviour-Klassen.

Objekt-Klassen-Instanziierung Für jedes XML-Objekt instanziiert die Facory ein GameObject und eine Objekt-Klasse, sogar wenn keine AssetBundles und Behaviour-Klassen angegeben sind. Die Instanz der Objekt-Klasse wird als Component an das GameObject gehangen. Das GameObjekt steht übergeordnet für das XML-Objekt.

AssetBundles Laden Unter den Umständen, dass in der XML-Objekt-Datei ein gültiger Pfad als Wert des <ModellPfad>-Tags eingetragen ist, liest die Factory das AssetBundle ein. Unity bietet hierfür die Funktion *Assetbundle.LoadFromFile*, welche im Konstruktor einen Pfad zu einem AssetBundle erwartet [Uni17,]. Die Factory instanziiert die in dem AssetBundle befindlichen Assets als GameObjects und ordnet sie in der Hierarchie unter dem GameObjekt an, dass stellvertretend für das XML-Objekt steht.

Behaviour-Instanziierung Anschließend erfolgt das Auslesen der Behaviour-Klassen. Unter dem <BehaviourListe>-Tag befinden sich alle zu instanziierenden Behaviours. Repräsentiert durch Elemente vom Typ des <Behaviour>-Tags. Um diese zu instanziieren, ist es essenziell in Erfahrung zu bringen, ob diese Werte für den Konstruktor übergeben werden. Andernfalls fände unter Verwendung eines ungültigen Konstruktors keine Instanziierung statt. Unter dem <WerteListe>-Tag befinden sich, falls angegeben, <WertObjekt>-Elemente. Jedes Element WertObjekt besitzt zwei Unterknoten. Einer gibt den Datentyp, der andere den Wert an. Die Zeichenkette der Werte wird ausgelesen und in ihren Datentyp geparst. Ist dies nicht möglich, erfolgt eine Weiterleitung an die Fehlerklasse (siehe 4.7). Alle für den Konstruktor einer Behaviour-Klasse benötigten Attribute werden in einem Array gespeichert. Die erste Stelle nimmt dabei immer die Objekt-Klasse ein, ihr folgen die Werte aus dem XML-Objekt. Dabei ist es wichtig, dass die Reihenfolge der Datentypen aus dem Attribute-Array, identisch mit denen des Konstruktors ist. Anschließend findet die Instanziierung der konkreten Behaviour-Klassen statt. Dafür bietet C#

die Funktion *Activator.CreateInstance*, diese instanziiert das konkrete Behaviour durch ihren Namen und die Konstruktorwerte aus dem Array, wie in Abbildung 4.7 zu sehen.

```
Behaviour CreateInstance(string klassenName,  System.Object[] werte, string XmlPfad){
    try{
        return (Behaviour)Activator.CreateInstance (Type.GetType(klassenName), werte);
    }catch(Exception e){
        FehlerAblauf.singelton.FalscheBehaviourKlasse (klassenName,XmlPfad,e.Message);
        return null;
    }
}
```

Abbildung 4.7.: Der Abgebildete C# Code instanziiert eine konkrete Behaviour-Klasse und übergibt ihr dabei benötigte Konstruktorwerte. Für den Fall eines Fehlers wird dieser an die FehlerKlasse weitergeleitet.

4.6.4. Die Pfad-Klasse

Aus Portierbarkeitsgründen ist es günstig, die Pfadreferenzen von Dateien relativ von ihrem zugehörigen Ordner im Wurzelverzeichnis aus zu Tätigen. Die Pfad-Klasse implementiert diese Funktionalität. Sie wandelt die relativen Pfadangaben in absolute um. Für jedes Verzeichnis mit referenzierbaren Inhalten stellt die Pfad Klasse eine Funktion zur Pfadumwandlung bereit. Damit diese Transformation geschehen kann, muss die Pfadklasse über den Ort des Wurzelverzeichnisses informiert werden (siehe Abbildung 4.8). Anschließend erzeugt sie von diesem ausgehend die absoluten Pfade zu den relativen Angaben. Ein Codebeispiel für die Implementierung ist in Abbildung 4.9 zu sehen.

Pfadangabe des Wurzelverzeichnisses sowie der XML-Startbedingung Um die Pfadangabe zum Wurzelverzeichnis sowie zur XML-Startbedingung zu ermöglichen, hat der Anwender der Software zu Programmstart die Möglichkeit, die Pfadangaben zu tätigen beziehungsweise sie zu verändern. Das Fenster zur Pfadangabe ist in Abbildung 4.10 zu sehen. Die Unity Engine ermöglicht durch die PlayerPrefs-Klasse kleinere primitive Daten persistent über Referenz eines Schlüssel zu hinterlegen [Uni17,]. Der eingegebene Pfad wird mittels dieser Klasse gespeichert und zur nächsten Ausführung geladen.

```
public void setPfad(string neuerPfad){
    if (!PfadPruefen (neuerPfad)) {
        FehlerAblauf.singelton.FalscherWurzelpfadPfad (neuerPfad);
        return;
    }
    pfad = neuerPfad;
}
```

Abbildung 4.8.: Der Abgebildete C# Code setzt den Pfad zum Wurzelverzeichnis innerhalb der Pfad Klasse, außerdem findet eine Überprüfung statt, ob es sich bei dem Pfad um einen existenten und erreichbaren Ordner handelt

```
public string ObjektXMLPfad(){
    return pfad + "Objekt_XML/";
}
```

Abbildung 4.9.: Die abgebildete, in C# implementierte, Funktion wandelt einen relativen XML-Objekt-Pfad in einen absoluten XML-Objekt-Pfad um.

4.7. Usability und Umsetzung

In diesem Abschnitt findet die Beschreibung der Implementierung von benutzerfreundlichen Funktionen des Toolkits statt. Die Usability-Verfahren sind im Abschnitt 3.7 erläutert.

Fehler Falls im Ablauf mit XML-Dateien und dem Lade- bzw. Instanziierungsvorgang Fehler auftreten, findet die Weiterleitung an die Fehlerklasse statt. Je nach Vorkommen des Fehler sind verschiedene Variablen nützlich, um diesen ausfindig zu machen. Die Klasse stellt also für verschiedene Fehlerarten Funktionen mit angepassten Parametern bereit. Über den Konstruktor der Funktionen übergibt das aufrufende Objekt Details zum Fehler und die Fehler-Klasse erstellt aus diesen eine Nachricht, welche das im Ablauf entstandene Problem wiedergibt sowie, falls es möglich ist, den Ort des Fehlers spezifiziert. Die Darstellung des Fehlers findet über eine Textausgabe auf dem Bildschirm statt, wie die Abbildung 4.11 zeigt.

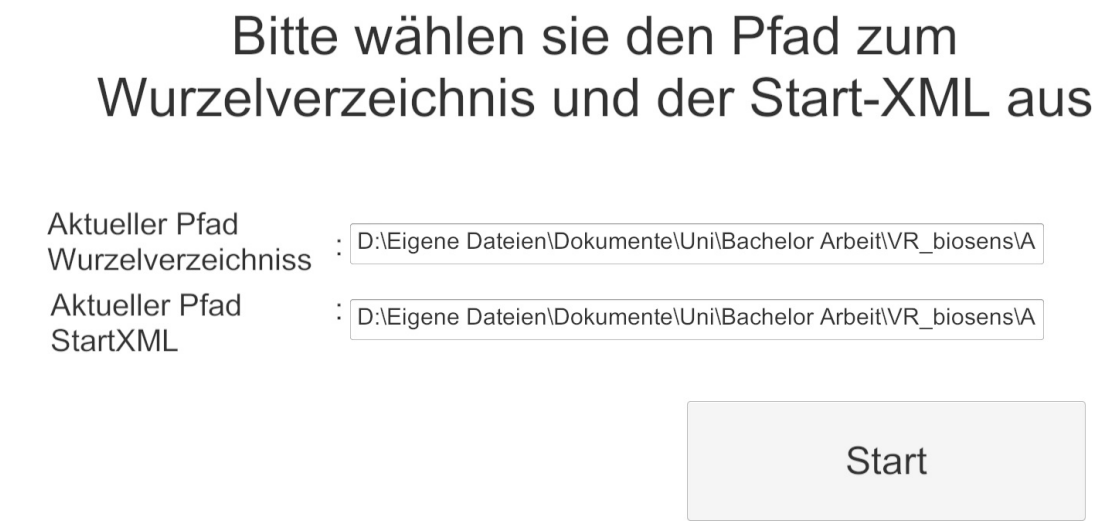


Abbildung 4.10.: Dieses Fenster erscheint zum Programmstart. Es gibt dem Nutzer die Möglichkeit Pfadangaben zu tätigen. Für den Wurzelverzeichnispfad und den XML-Startbedingungspfad existieren separate Eingabefelder.

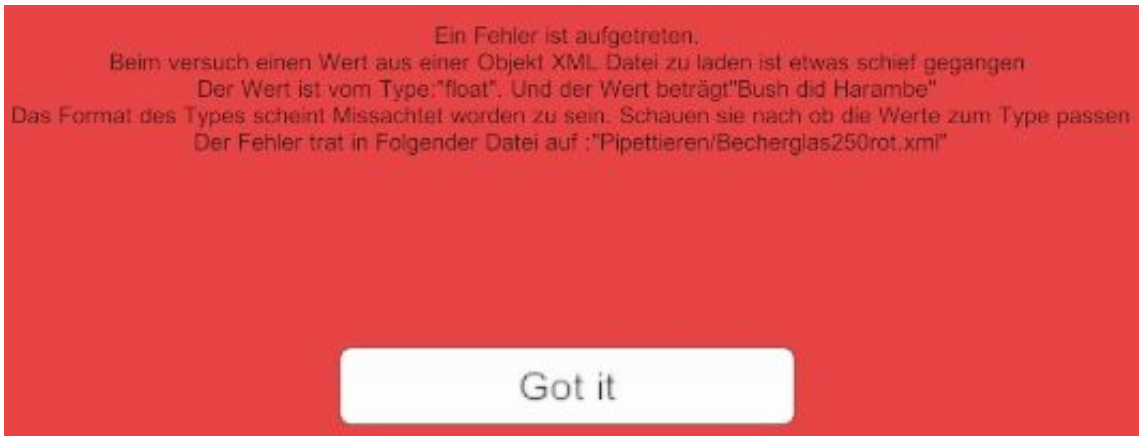


Abbildung 4.11.: Eine Fehlernachricht entstanden durch einen Wert der nicht zu angegebenen Werttyp passt.

Haptik des Vive Controllers Der Vive Controller bietet die Möglichkeit ein haptisches Feedback zu erzeugen. Dafür existiert eine bereitgestellte Funktion TriggerHapticPulse [Uni17,]. Diese sorgt dafür, dass der Vive Controller vibriert.

Optisches Feedback Das optische Feedback besteht aus der Färbung von Objekten wie in 3.7 beschrieben. Wenn der Benutzer die Pipette nutzt, behält sie ihre ursprüngliche Materialfarbe bei. In Abbildung 4.12 ist dieser Zustand zu sehen. Befindet sich aber eine Pipette nahe genug am Pipettenhalter, so löst sie die Veränderung der Farbe des Pipettenmaterials in grün aus, wie in Abbildung 4.13 zu sehen ist. Wird sie nun losgelassen, so wird das Material wieder in den Ausganszustand versetzt und die Pipette an den Halter positioniert, verdeutlicht in Abbildung 4.14 .

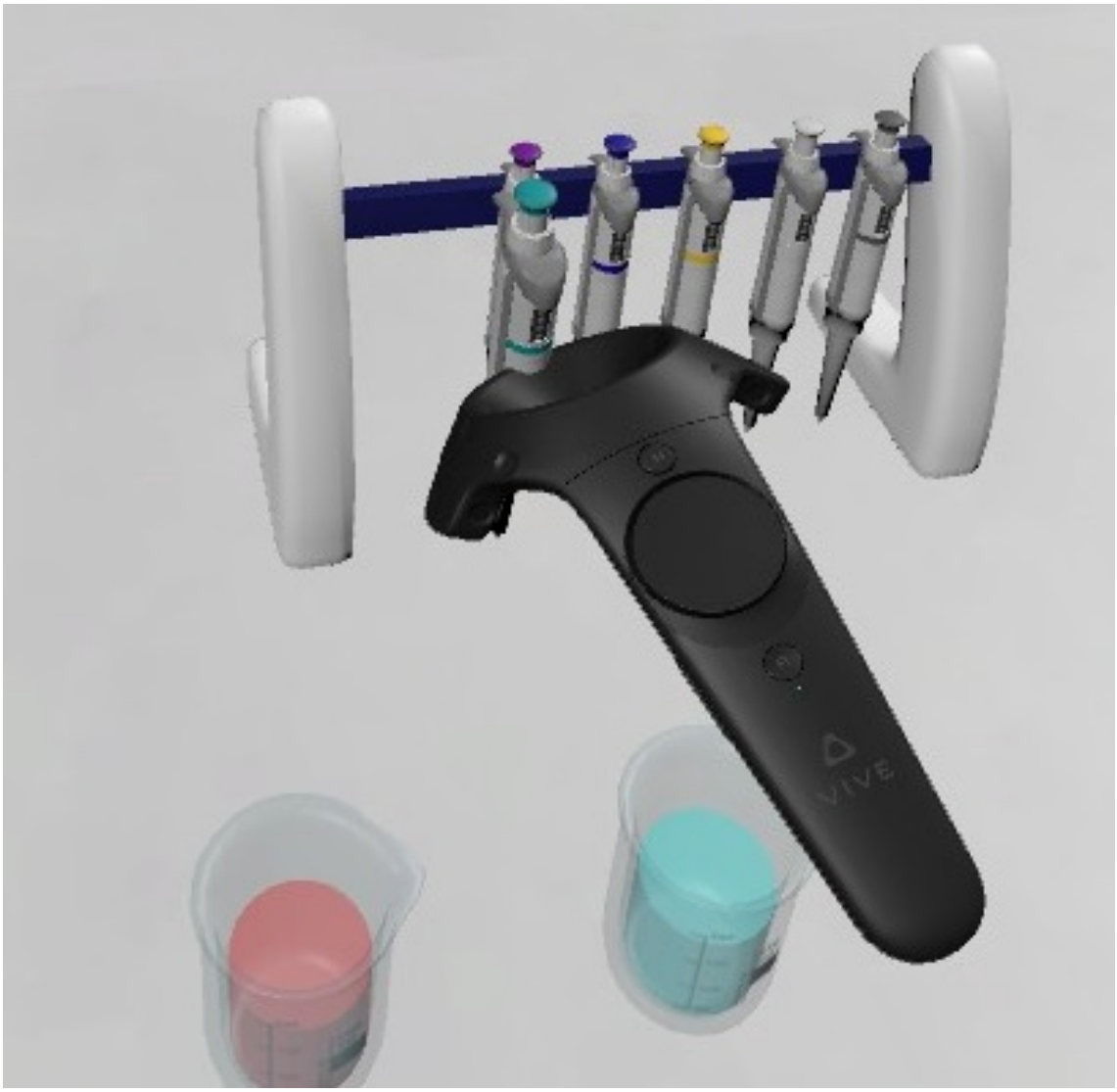


Abbildung 4.12.: Die Pipette befindet sich noch nicht im Bereich, um sie zurück an die Halterung anzubringen. Ihre Farbe verbleibt unverändert.

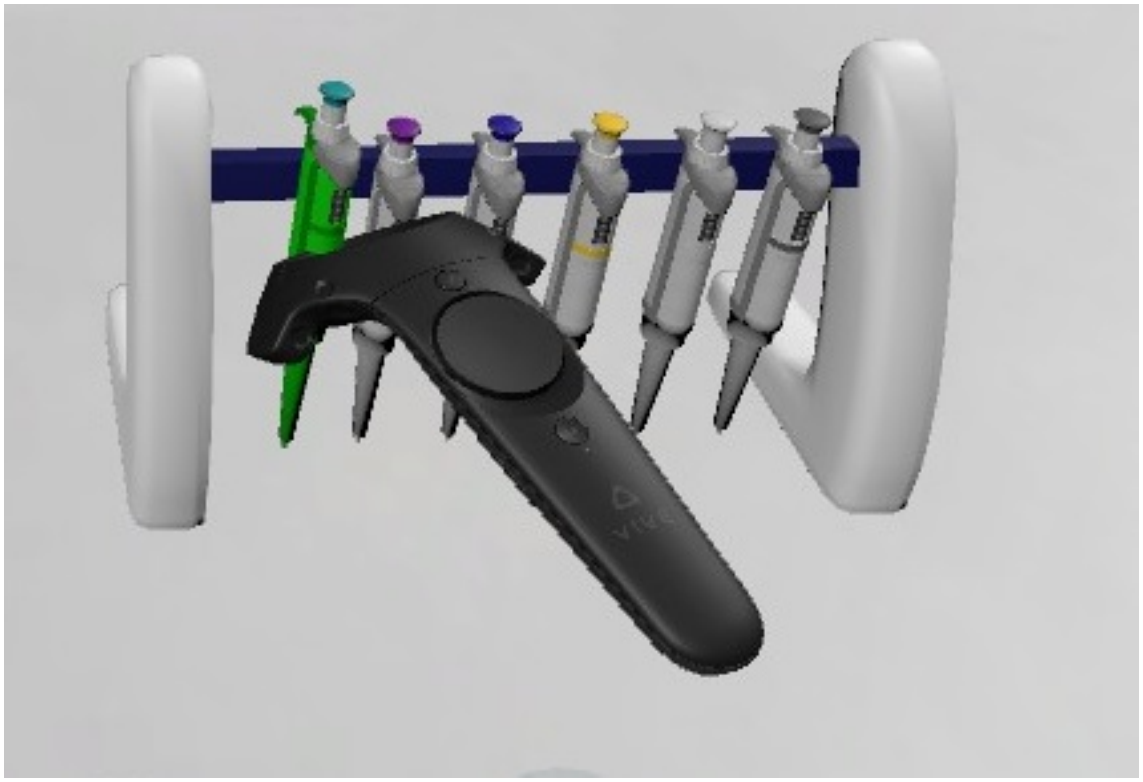


Abbildung 4.13.: Die Pipette hat den Bereich erreicht in welchen sie, wenn sie losgelassen wird, an den Pipettenhalter positioniert wird. Die grüne Farbe indiziert diesen Sachverhalt.

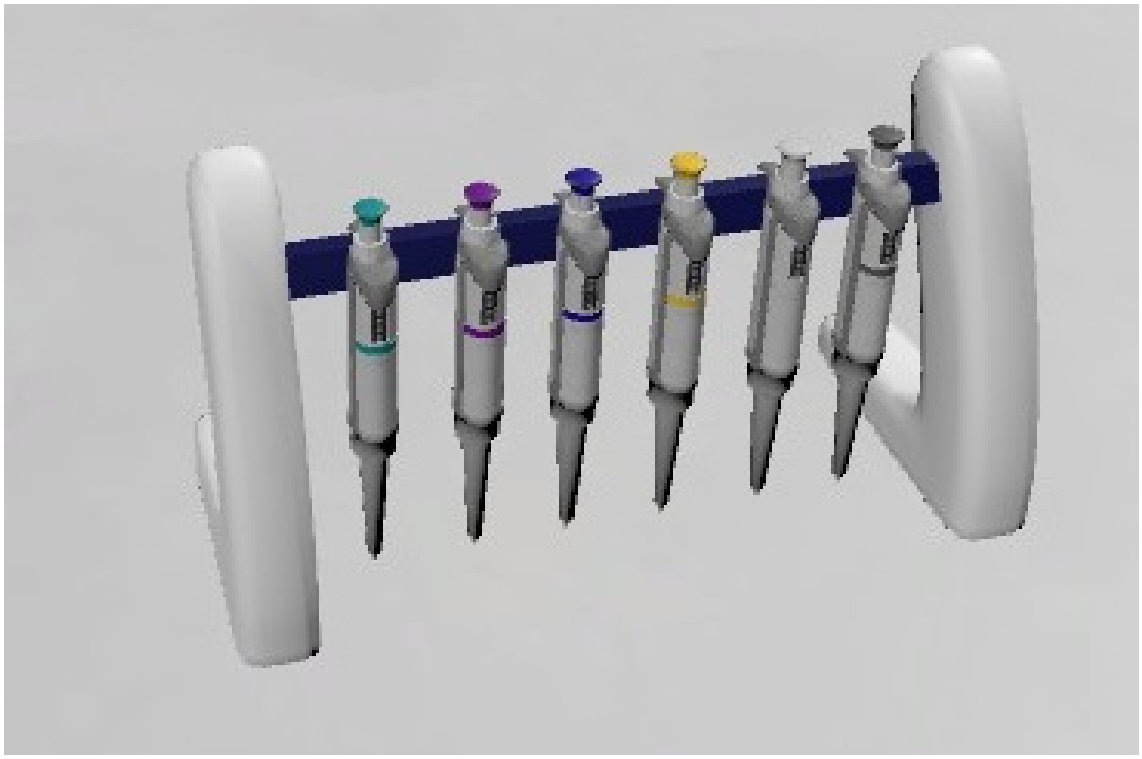


Abbildung 4.14.: Die Pipette befindet sich nun wieder am Pipettenhalter und auch die Materialfarbe hat ihren Ausganszustand eingenommen

5. Evaluation

Dieses Kapitel befasst sich mit der entstandenen Software. Dabei wird auf den Nutzen sowie auf die Vor- und Nachteile des Systems eingegangen. Die benutzerfreundlichen Methoden, sowie entstandene Probleme werden ausgewertet.

5.1. Versuch Bedienbarkeit des Systems

Das entstandene System erfüllt das im Kapitel 1 definierte Ziel der Arbeit. AssetBundles und Behaviour-Klassen können über dieses geladen werden. Ein Versuch soll prüfen, ob es einen Probanden ohne nennenswerte Informatikkenntnissen möglich ist, das System zu nutzen.

5.1.1. Versuchsaufbau

Ein Versuchsaufbau wurde definiert. Der Proband hat die Aufgabe dem System eine neue XML-Objekt-Datei zuzuführen und diese anschließend durch die Software laden zu lassen. Dazu erhält er einen Build des in dieser Arbeit entstandenen Projektes, sowie einen Wurzelverzeichnis-Ordner und eine Anleitung im PDF-Format. In der Anleitung wird erklärt, wie ein extra bereitgestelltes AssetBundle durch eine neue XML-Objekt-Datei zusammen mit einer Behaviour-Klasse geladen wird (siehe A.1.1). Der Proband soll die Aufgaben mittels der Anleitung ohne Hilfe von außen lösen. Dieser Versuch findet in 2 Durchgängen statt, um zu ermitteln, wie lange es dauert das Softwaresystem ohne menschliche Hilfe zu bedienen. Des weiteren ergibt sich aus der Zeit des zweiten Durchlaufes, wie gut sich der Proband in die Funktionsweise des Systems eingefunden hat. Der Proband bleibt hierbei anonym. Der Versuch findet auf dem in Tabelle 5.1 spezifizierten Computer statt.

Systeminformationen	
Betriebssystem	Windows 10 Enterprise 2016 LTSB 64-Bit-Version (10.0, Build 14393)
Prozessor	Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz
Speicher	32768MB RAM
Grafikkarte	NIVIDA GeForce 1080 8 GB GDDR5X

Tabelle 5.1.: Systeminformationen

5.1.2. Ergebnis des Versuchs

Der Proband hat das System in beiden Durchläufen erfolgreich bedienen können. Dabei hat der erste Durchlauf 18 Minuten und 13 Sekunden in Anspruch genommen. Dabei ist zu berücksichtigen das dieser Durchlauf den ersten Kontakt mit dem System darstellt und zusätzliche Zeit für das komplette Lesen der Anleitung benötigt wurde. Der zweite Durchlauf konnte in nur 4 Minuten und 28 Sekunden ausgeführt werden.

5.1.3. Auswertung des Versuches

Der Versuch zeigt, dass es einem Anwender ohne spezielle Informatikkenntnisse möglich ist, das Softwaresystem zu bedienen. Nach bereits einem Versuch fängt dieser an, sich in das System einzufinden. Die Anleitung hilft, das Verständnis für die Arbeit mit dem Softwaresystem zu prägen.

5.2. Aufgetretene Probleme

Während der Entwicklung und Implementierung sind Probleme aufgetreten. Wie diese gelöst wurden wird in diesem Abschnitt erläutert.

Wiederholendes laden eines AssetBundles Ursprünglich sollten AssetBundles jedes mal geladen werden, wenn sie in einer zu ladenden XML-Objekt-Datei vorkommen. Dabei ist es vorgekommen das AssetBundles mehrfach auftauchten (beispielsweise wenn eine XML-Objekt-Datei mehrfach geladen wird). Dies führte zu

einen Fehler innerhalb der Unity Engine-Schnittstelle und damit zum Ausfall des Systems. Als anfängliche Problemumgehung wurden AssetBundles nach dem Laden und Instanziieren wieder gelöscht. Die Softwareplattform war danach wieder ausführbar, litt jedoch unter teilweise fehlenden Texturen. Der Grund für dieses Verhalten ist unbekannt. Gelöst wurde das Problem schließlich durch ein Dictionary (dt. Wörterbuch). Ein Dictionary ist ein Konstrukt in der Informatik, welches Objekte und Schlüssel, bestehend aus Zeichenketten, zusammen koppelt. Immer wenn ein AssetBundle geladen werden soll, prüft die Software, ob sich dessen Pfad im Dictionary befindet. Ist dem nicht so, lädt das Softwaresystem das AssetBundle und fügt dieses dem Dictionary hinzu. Die Assets werden instanziiert und der der Pfad wird als Schlüssel im Dictionary angegeben. Ist der Pfad bereits im Dictionary vorhanden, so findet dieses das zugehörige AssetBundle. Die in diesem befindlichen Assets gilt es anschließend wie gewohnt zu instanziieren.

Positionierung neuer Objekte Beim Laden neuer AssetBundles und XML-Objekt-Dateien entstand das Problem der Positionierung. Nutzer sollen ohne Nutzung der Unity Engine in der Lage sein AssetBundles zu laden, können aber schlecht abschätzen, welche Position in der XML-Startbedingung angegeben werden muss damit die Instanziierung an der gewünschten Stelle stattfindet. Als Lösung des Problems kommt der Positionswürfel zum Einsatz. Der Positionswürfel ist ein interagierbarer Würfel, der seine aktuelle Position anzeigt (siehe 5.1). Er kann mittels des Controllers aufgenommen und neu positioniert werden. Dadurch entsteht die Möglichkeit, Objekte ohne Hilfe der Unity Engine an gewünschten Stellen zu positionieren.

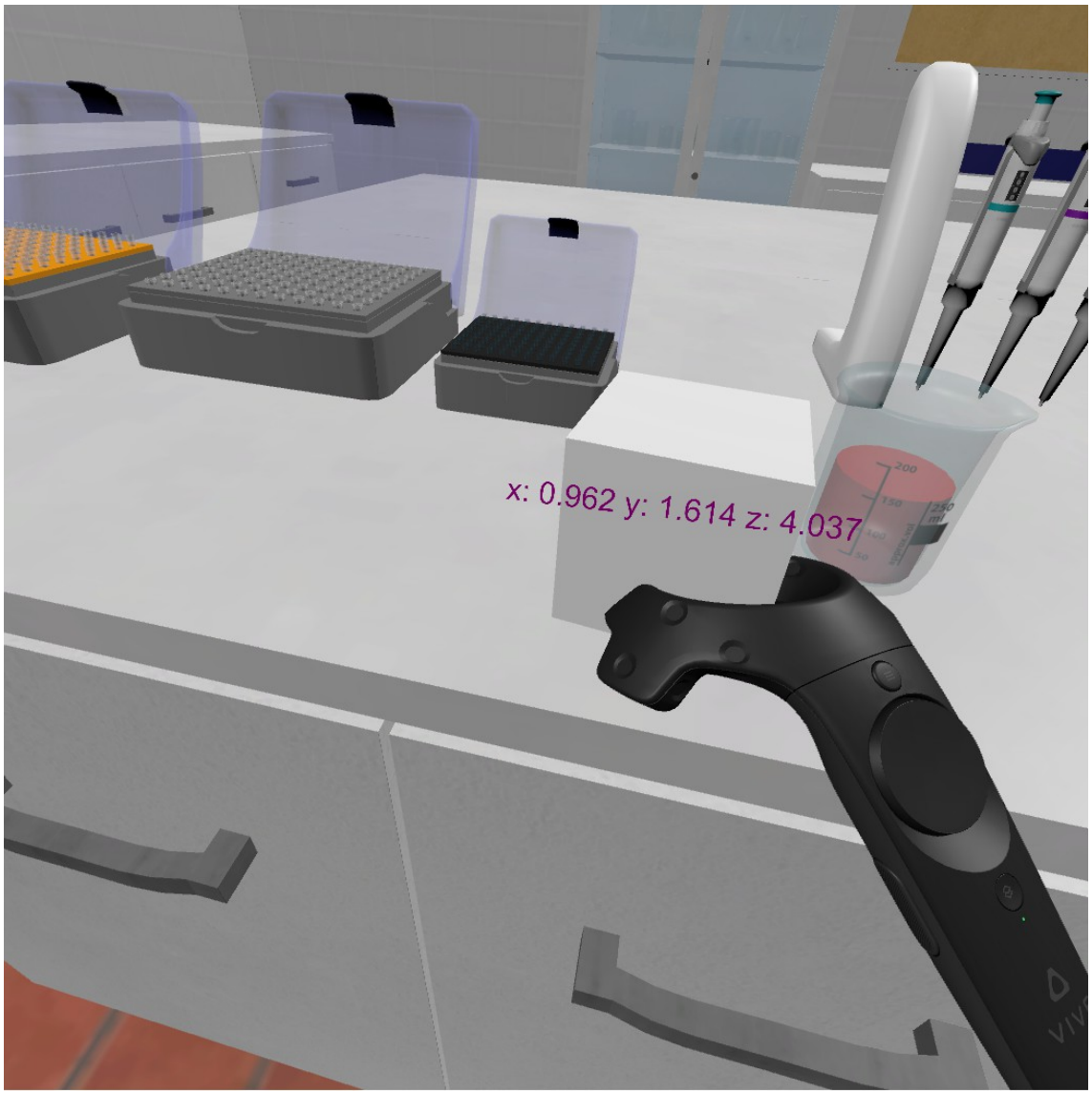


Abbildung 5.1.: Der Postionswürfel zeigt seine Position im Unity-World-Space an [Uni17,].

6. Zusammenfassung und Ausblick

Eine Zusammenfassung, sowie der Ausblick auf Erweiterungsmöglichkeiten sind Inhalt dieses Kapitels. Ziel der Arbeit ist es, wie in Kapitel 1 beschrieben, eine Softwareplattform zu implementieren, die es einer Unity-Anwendung ermöglicht, Inhalte konfigurier- und erweiterbar zu gestalten. Die im Kapitel 2 gesammelten Informationen bilden die Grundlage für den weiteren Verlauf und die Konzeptionierung im Kapitel 3. Dort findet die Spezifizierung des Softwaresystems statt. Die Evaluation der Funktionalität erfolgt im Kapitel 5.

Als Ergebnis der Arbeit ist ein System entstanden, das die erweiterte Funktionalität des dynamischen, konfigurierbaren modularen Ladens zur Laufzeit realisiert. Des weiteren wurden nutzerfreundliche Konzepte für den Anwendungsfall des Pipettierens umgesetzt.

6.1. Ausblick

Für das entstandene Softwaresystem sind Erweiterungsmöglichkeiten vorhanden. Nutzerfreundliche Rückmeldungen finden aktuell auf optisch- und haptischen Kanälen statt. Eine Erweiterungsmöglichkeit besteht in der zusätzlichen Unterstützung von akustischen Signalen als Rückmeldung.

Arbeit mit XML-Dateien finden über einen Texteditor statt. Hierbei ist die Gefahr der Erstellung fehlerhafter Inhalte groß. Die Erweiterung durch einen visuellen XML-Editor bringt Vorteile. Ein solcher könnte als Funktionalität Kenntnisse über Dateien des Wurzelverzeichnisses sammeln und bereitstellen. Dadurch müssten Anwender keine XML-Dateien ändern, sondern können über ein User-Interface alle nötigen Konfigurationen vornehmen.

6.2. Fazit

Das Softwaresystem erfüllt die Anforderungen des dynamischen und konfigurierbaren Ladens von Modulen in Form von AssetBundles. In Kooperation mit Kerstin Knura (Hochschule Mittweida) ist eine konfigurierbare Laborumgebung für die Simulation des Pipettiervorgangs entstanden. Diese ist zudem änder- und erweiterbar.

Literaturverzeichnis

[Bur03] Grigore C. Burdea: *Virtual Reality Technology*, John Wiley Sons Inc., 2 Aufl., 2003.

[Cry17] Crytek: *CRYENGINE V Manual*, 1 Aufl., 2017, URL: http://docs.cryengine.com/display/CEMANUAL/CRYENGINE+V+Manual abgerufen am 23.11.2017.

[Epi17] Epic Games: *Unreal Engine Documentation*, 1 Aufl., 2017, URL: https://docs.unrealengine.com/latest/INT/Engine/ abgerufen am 25.11.2017.

[GHJV15] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides: *Design Patterns*, mitp, 1 Aufl., 2015.

[Gol14] Joachim Goll: *Architektur- und Entwurfsmuster der Softwaretechnik*, Springer, 1 Aufl., 2014.

[Gre14] Jason Gregory: *Game Engine Architecture*, CRC PRess, 2 Aufl., 2014.

[Jos08] Bipin Joshi: *Beginning XML with C# 2008*, Apress, 2 Aufl., 2008.

[Kru17] Kerstin Kruna: *Entwicklung und Implementierung eines virtuellen Labors zur Durchführung biochemischer Experimente am Beispiel einer Verdünnungsreihe mittels Pipettieren*, Hochschule Mittweida, 2017, bachelor Thesis.

[Mic17] Microsoft: *CSharp Documentation*, 1 Aufl., 2017, URL: https://docs.microsoft.com abgerufen am 28.11.2017.

[Moo11] Michael E. Moore: *Basics of game design*, CRC Press, 1 Aufl., 2011.

[Oes01] Bernd Oestereich: *Objektorientierte Softwareentwicklung Analyse und Design mit der Unified Modeling Language*, Oldenburg, 5 Aufl., 2001.

[RR03] Gunther Rothfuss und Christian Ried: *Content Management mit Xml*, Springer, 2 Aufl., 2003.

[Uni17] Unity Technologies: *Unity User Manual*, 1 Aufl., 2017, URL: https://docs.unity3d.com/Manual/index.html abgerufen am 25.11.2017.

[VIV17] HTC VIVE: *Htc Vive Specifications*, 2017, URL: https://www.vive.com/de/product/#vive-spec, besucht am 23.11.2017.

Anhang

Anhangsverzeichnis

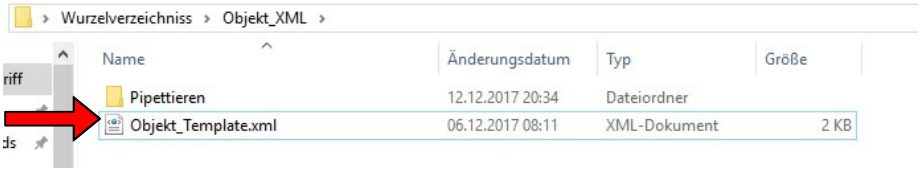
Anhang - Abbildungsverzeichnis

A. Analysedokumente

A.1. Anleitung XML-Objekt Erstellung

Anlegen und verwenden einer XML-Objekt Datei

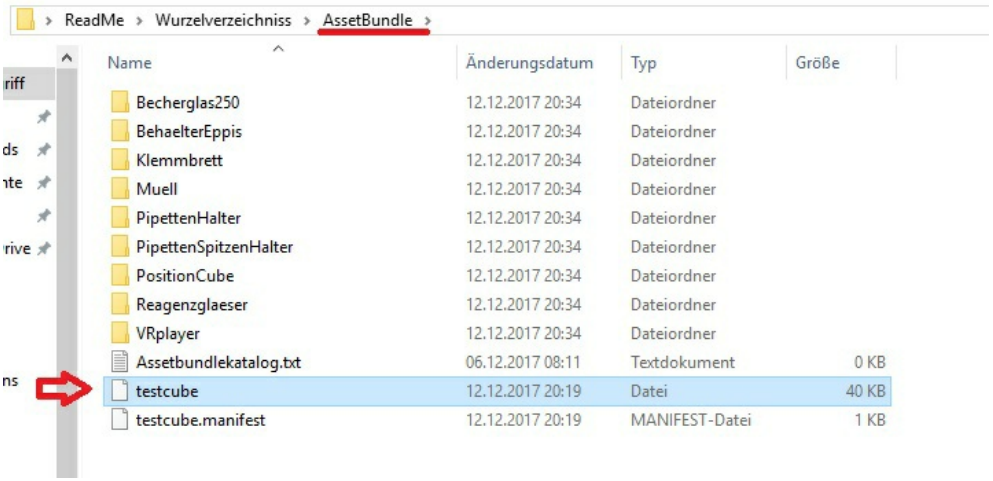
Im Ordner Objekt_XML befindet sich eine Datei namens Objekt_Template. Diese beinhaltet die Grundstruktur einer XML-Objekt-Datei



Um eine neue XML-Objekt-Datei anzulegen wird die Objekt_template.xml Datei einfach kopiert und mit einen neuen Namen versehen

Nun haben wir eine neue XML-Objekt Datei erstellt. In dieser Können nun AssetBundles und Behaviour Klassen angegeben werden

Nun wollen wir ein AssetBundle angeben. Dazu gilt es herauszufinden wo sich dieses befindet und wie es heißt



Das AssetBundle besitzt keine Dateiendung. Der Pfad zu einem Assetbundle geht immer vom AssetBundle Ordner aus.
Im diesen Fall ist der Pfad einfach nur testcube. Würde sich der testcube beispielsweise unter einem Ordner befinden, nennen wir ihn Test so wäre der Pfad Test/testcube

Dieser Pfad muss nun in der frisch angelegten XML-Objekt-Datei angegeben werden. Öffnen wir die Datei mit einen Texteditor, in diesem Beispiel wird Notepad++ benutzt

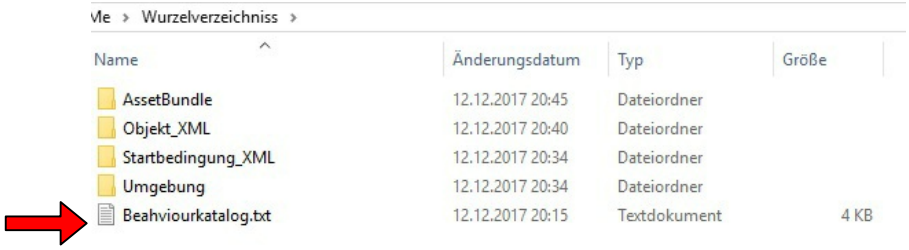
```
<!-- DerAssetBundle-Pfad ist der Pfad zu einem AssetBundle innheralb des AssetBundleOrdners -->
<AssetBundle></AssetBundle>
```

Hier ist eine Abbildung des AssetBundle Tags. Zwischen das öffnende <> und schließende </> Tag muss jetzt der Pfad angegeben werden

```
<!-- Der AssetBundle-Pfad ist der Pfad zu einem AssetBundle innheralb des AssetBundleOrdners -->
<AssetBundle>testcube</AssetBundle>
```

Jetzt weiß das XML-Objekt welches Assetbundle zum Programmstart geladen werden soll

Anschießend soll dem AssetBundle eine Behaviour Klasse zugeteilt werden. Behaviour Klassen sind spezielle Funktionen. Dazu muss im Behaviour.Katalog geschaut werden welche Behaviour-Klassen existieren



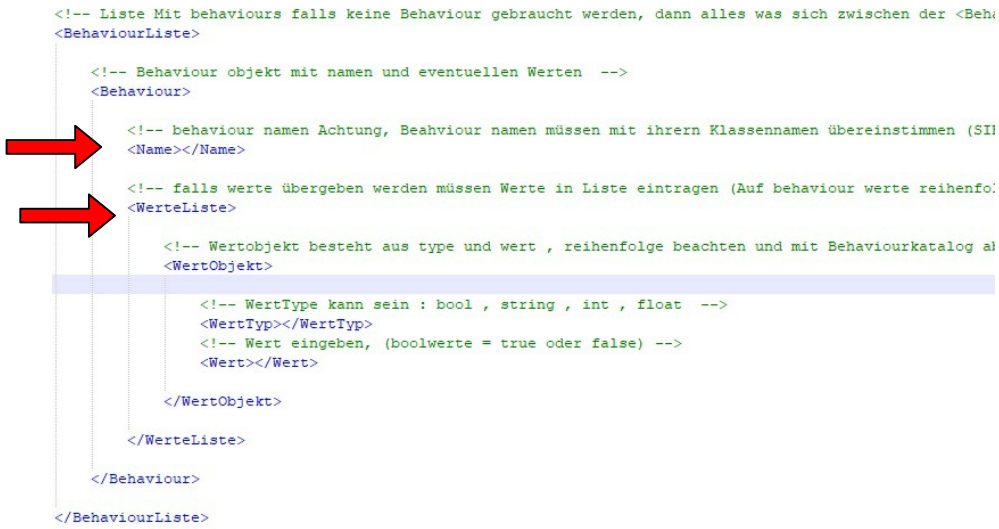
Wir wollen das SteuerbarBehaviour laden. Dazu suchen wir das SteuerbarBehaviour im Katalog

```
SteuerbarBehaviour                                                    ###     ()
### Abhängigkeiten : Das Asset aus dem AssetBundle benötigt einen Rigidbody und einen Collider
### beschreibung   : Das SteuerbarBehaviour-Behaviour macht Assets durch den Vive Controller interagierbar.
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
```

Um ein Behaviour zu nutzen müssen zwei Dinge beachtet werden.

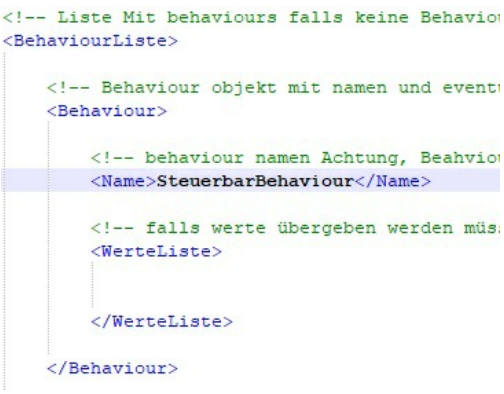
1. Der Name des Behaviours
2. Die Werte die das Behaviour benötigt

In diesem Fall ist der Name "Steuerbarbehaviour" und das Behaviour benötigt keine Werte. (siehe übernächste Abb.) Diese Informationen gilt es in das XMl-Objekt zu schreiben

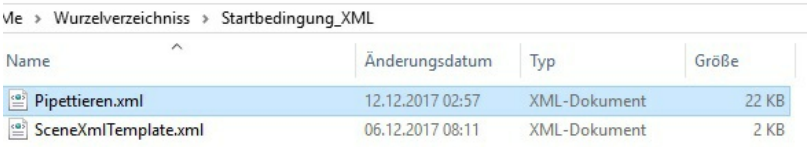


In das Name Tag muss der Namen des Behaviours geschrieben werden.
Werte müssen für diese Behaviour nicht angegeben werden, wir können die Werteliste Löschen.
Nicht teil der Anweisungen, kann übersprungen werden :
Anmerkungen für Nutzer die Mehrere Behaviour-Klassen oder Werte angeben wollen:
Um mehrer Behaviour-Klassen anzugeben müssen mehrere Behaviour Tags angelegt werden

Werte Werden im WertObjekt angegeben sie bestehen aus Typ und Wert (siehe Behaviour Katalog) WertObjekte können als Liste strukturiert werden. damit es möglich ist mehrere anzugeben.



Nun haben wir ein XML-Objekt angelegt und gefüllt. Nun müssen wir das Programm noch informieren, dass es auch genutzt werden soll. Dazu muss das XML-Objekt in der Startbedingung-XML angegeben werden. Diese befindet sich im Startbedingung XML-Ordner und heißt Pipettieren.xml



Diese öffnen wir wieder mit dem Texteditor (hier Notepad++). Die XML-Startbedingung enthält Auskunft welche XML-Objekt geladen werden sollen und an welcher Position dies geschehen soll

Abbildung A.1.1.: PDF-Anleitung, welche dem Nutzer durch Abbildungen und Texte erläutert wie eine neue XML-Objekt-Datei erstellt, konfiguriert und eingefügt wird.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendete Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Mittweida, den 14. Dezember 2017

Gabor Schulze